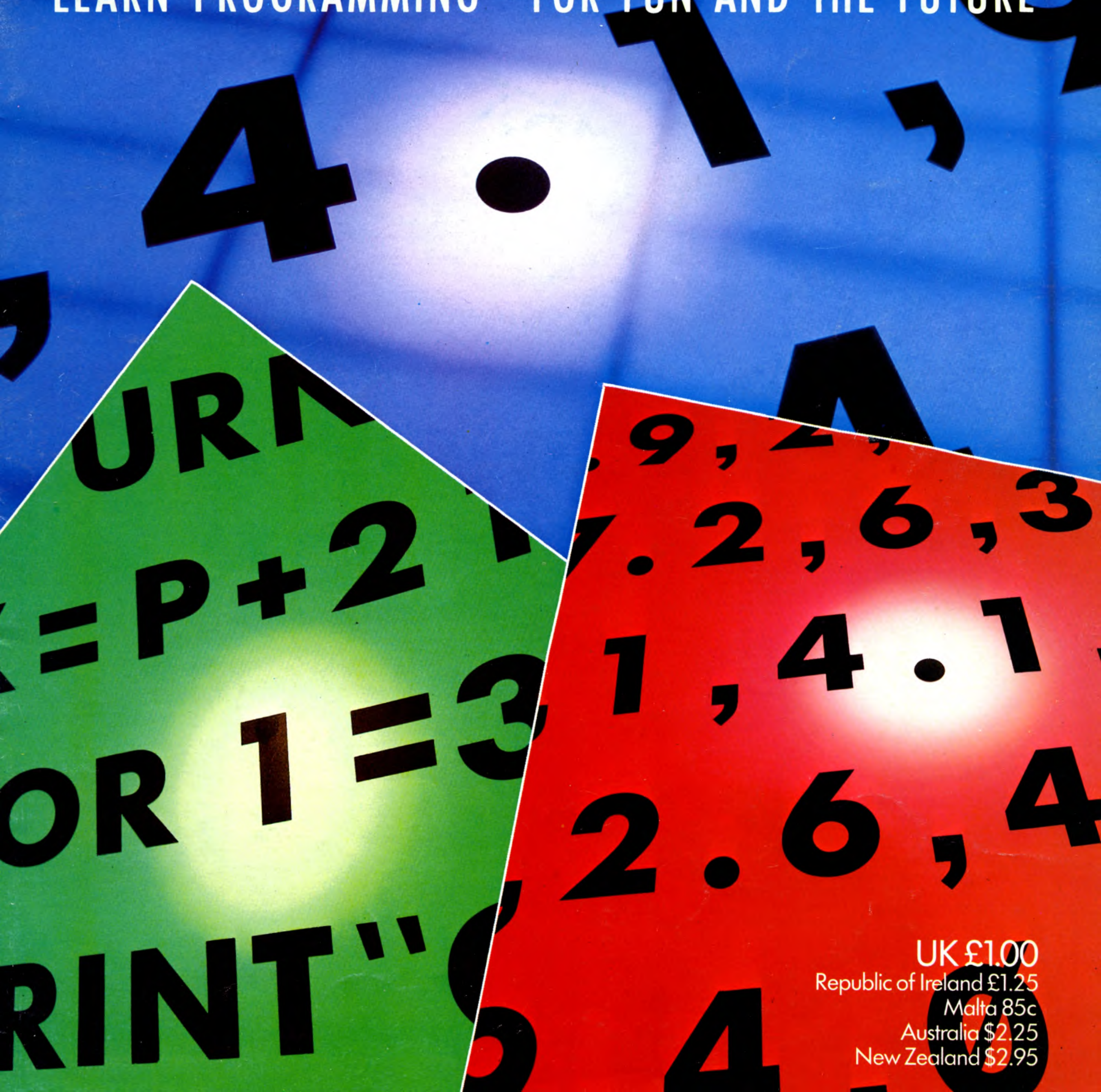
A MARSHALL CAVENDISH 11 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 1 **No 11**

PICTURE CREDITS
Front cover, Howard Kingsnorth. Pages 321, 322, 324, 326, Peter Richardson. Pages 328, 330, Kuo Kang Chen. Page 331, Nick Farmer. Page 333, Malcolm Livingstone. Page 334, Tudor Art Studios. Page 336, Bernard Fallon. Pages 337, 338, Howard Kingsnorth. Page 339, Paul Chave. Page 340, Mick Saunders. Page 343, Peter Western. Pages 344, 346, 347, 348, Alan Baker. Page 350, Jon Stewart/Roy Flukes. Page 351, Bernard Fallon.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +

COMMODORE 64 and 128

ACORN ELECTRON, BBC B and B+

DRAGON 32 and 64

ZX81

VIC 20

TANDY TRS80 COLOUR COMPUTER

ASSEMBLING BY HAND-2

- PRACTISE YOUR ASSEMBLY
- A ROUTINE TO SCROLL THE SCREEN SIDEWAYS
- UNDERSTANDING HOW THE PROGRAM WORKS

Once you have learnt to assemble by hand, the world of home computing is your oyster. You should already have a knowledge of BASIC, you can handle machine code and translating assembly language into machine code will be no problem

In the first part of this article on hand assembly you learnt how to convert assembly language into machine code so that you could enter it into your computer and execute it.

With this mastered, you have a chance to try out these techniques on some useful routines.

SIDEWAYS SCROLLING

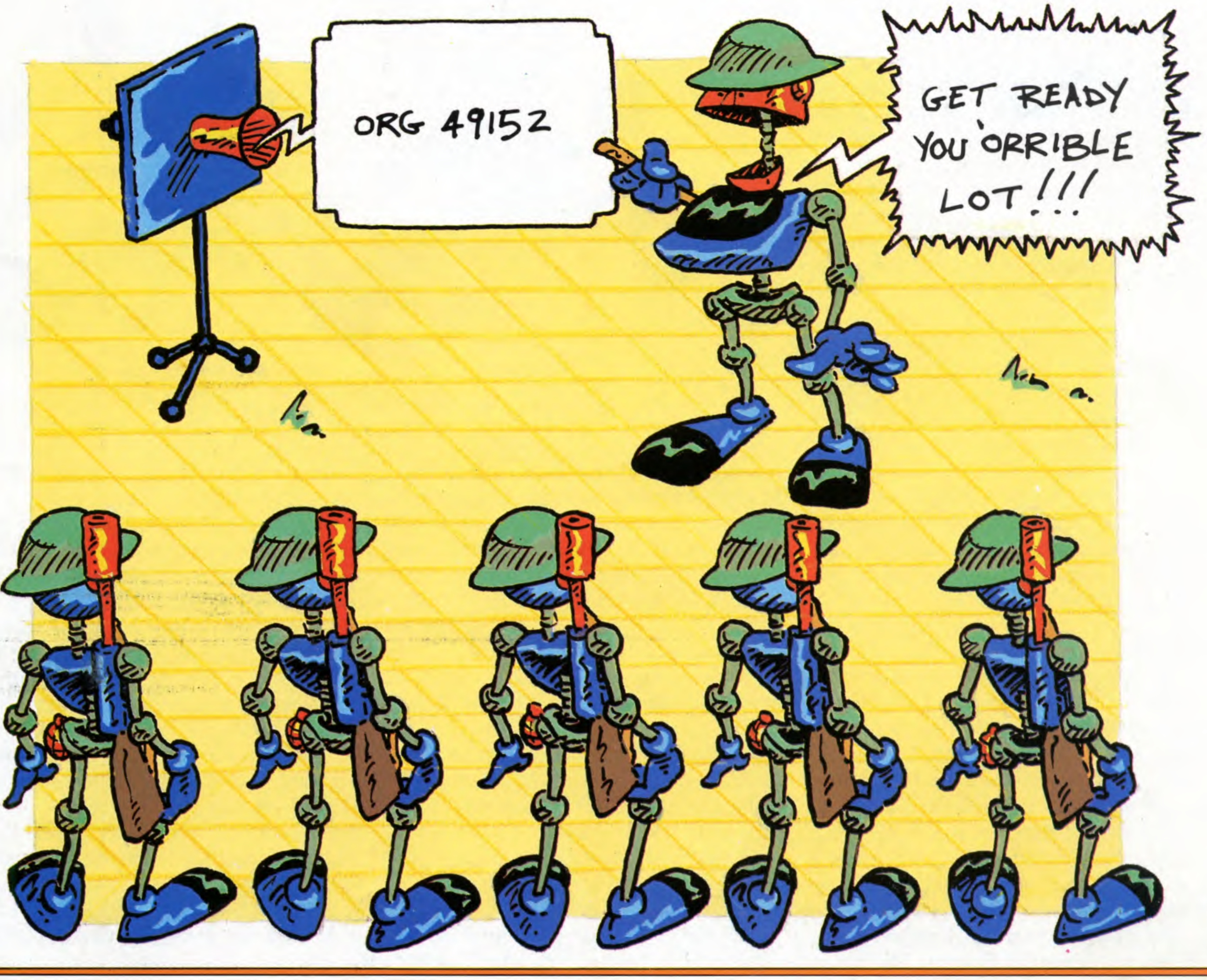
The following assembly language programs scroll the screen to the left and to the right one character square. Hand-assemble them and feed the machine code into your computer using your machine code monitor program. If you want to save these programs for use in the future don't forget to put them into different parts of memory in case you want to use them both at the same time.

Both programs work independently of their position in memory so you can put them anywhere convenient on your machine.

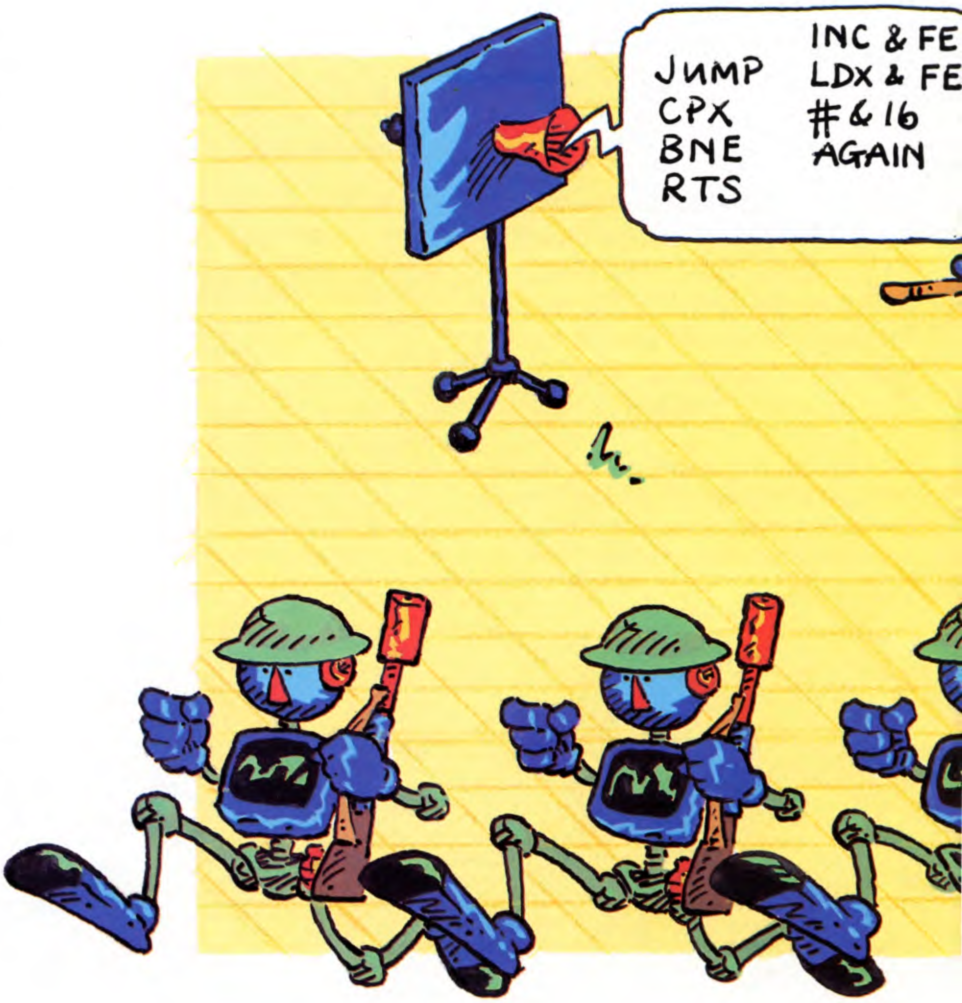
If you want the screen to scroll to the left or to the right by more than one character square you can call the machine code routine out of a FOR ... NEXT loop. Or to make it scroll just when you press a key, call it out of a BASIC program using INKEY$ or GET$ (see page 132).

S

This routine scrolls the screen to the left. Here it has been translated into machine code.



```
     ld de,16384           11 ØØ 4Ø
     ld hl,16385           21 Ø1 4Ø
     ld b,192                 Ø6 CØ
loop push bc                     C5
     ld a,(de)                   1A
     ld bc,31              Ø1 1F ØØ
     ldir                     ED BØ
     ld (de),a                   12
     inc de                      13
     inc hl                      23
     pop bc                      C1
     djnz loop                1Ø F3
     ret                         C9
```



HOW IT WORKS

The **ld de,16384** and **ld hl,16385** load the addresses of the first two bytes of the Spectrum's screen display file into the DE and HL registers. The **ld b,192** loads the B register with the number 192—there are 192 lines on the Spectrum's screen and the B register is to be used as a counter to count off the lines.

The problem is the B register is going to be used for other things as well, so for the moment, its contents are going to be stored on the stack. But the stack command won't take the B register on its own, only the B and C registers together. So **push bc** is used. It doesn't matter what is in the C register at this point because, in this operation, we are not going to look at the C register.

The **ld a,(de)** loads the accumulator with the contents of the memory location whose address is in the DE register—16384, remember. This is an example of indirect addressing.

The **ld bc,31** loads the BC register with the number 31. There are 32 character squares in a line but shifting the first one into the last position is done independently. So for this operation you only need to count up to 31.

The crucial instruction here is **ldir**. This means **l**oa**d**, **i**ncrement and **r**epeat. What it does is load the contents of the memory location whose address is in the HL register—16385 on the first pass—into the memory location whose address is contained in DE—16384 on the first pass. Then it increments HL and DE, decrements BC, checks if the BC is zero—if it is not the whole instruction is repeated.

The effect of this is to move what is in the second screen location into the first, then the third into the second, and so on all the way along the first row. It stops when the BC register has counted down from 31 to Ø. The last operation it does is move the 32nd screen location into the 31st position. But then when the BC register is decremented, it becomes Ø and the program moves on to the next instruction.

The **ld (de),a** puts the contents of the accumulator into the address pointed to by the DE register. Looking back you will see that the accumulator contains the contents of the DE register before the **ldir** instruction started incrementing it—in other words the contents of 16384, the first screen location.

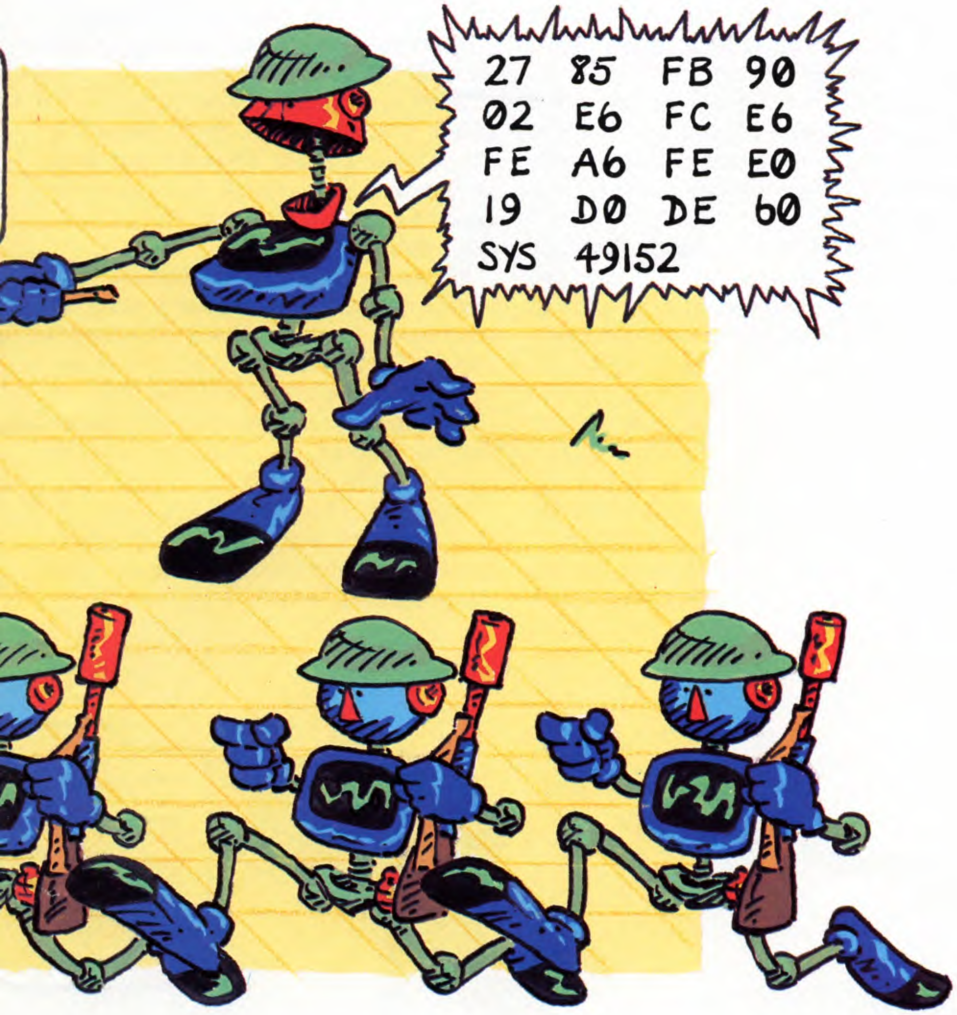
The **DE** register has been repeatedly incremented since then though. It now points to the last screen location—the exact position you want to slot the contents of the first screen location into.

The **inc hl** and **inc de** increment the HL and the DE registers so they are set up at the beginning of the next line. As you swapped the contents of the first location into the last by hand (as it were) you have to increment these two registers yourself, rather than allow the **ldir** instruction to do it for you.

The **pop bc** pulls the contents of the two memory addresses at the top of the stack and puts them back into the B and C registers. **djnz** **d**ecrements the B register and **j**umps if its contents are **n**ot **z**ero. At the beginning of the first pass you loaded the B register with 192. This was then stored on the stack while you did other things with the B and C registers. The **djnz** decrements the B register to 191 after the first pass. This is not zero, so the instruction jumps back to the first appearance of the label it's been given, **loop**.

This jump will occur over and over again while the instruction is counting down from 192 to Ø—dealing with a line of the screen at a time. When the B register does become Ø after it has been decremented by the **djnz** instruction the jump is not made and the program goes on to the next instruction.

The **ret** means **ret**urn. No machine code



subroutine will run without a return instruction, otherwise it will not return to the BASIC. The microprocessor will then plough on into any rubbish that follows your machine code program in memory. The chances are that the program will then crash and memory will be corrupted.

Even if this doesn't happen, the microprocessor will career on out of the top of memory and back into the bottom. There it will hit the initializing routines which will clear out the memory anyway. So remember the importance of **ret**.

Now try hand assembling the routine which follows to scroll the screen to the right yourself.

```
     ld de,22527
     ld hl,22526
     ld b,192
loop push bc
     ld a,(de)
     ld bc,31
     lddr
     ld (de),a
     dec hl
     dec de
     pop bc
     djnz loop
     ret
```

This program, you notice, starts at the end of the screen display file, and works backwards through it. The only other thing that is different from the previous program is the **lddr** instruction. This means **l**oad, **d**ecrement and **r**epeat. This loads the contents of the memory location whose address is contained in HL into the memory location whose address is in DE again. But then it decrements HL and DE. It decrements BC, like the **ldir** instruction, and again repeats if the BC registers do not contain Ø.

The jump is the same length as before. So you can check you've worked the label out correctly by looking back at the machine code listing above. (Remember to count the bytes from the end of the jump instruction, that is including the jump byte you are working out.)

This assembly language routine scrolls the screen to the left on any ZX81 which has an 8K or 16K RAM pack attached. Here it has been translated into machine code for you.

```
     LD DE,(16396)        ED 5B ØC 4Ø
     INC DE                         13
     LD H,D                         62
     LD L,E                         6B
     INC HL                         23
     LD B,24                     Ø6 18
LOOP PUSH BC                        C5
     LD A,(DE)                      1A
     LD BC,31                 Ø1 IF ØØ
     LDIR                        ED BØ
     LD (DE),A                      12
     INC HL                         23
     INC HL                         23
     INC DE                         13
     INC DE                         13
     POP BC                         C1
     DJNZ LOOP                   1Ø F1
     RET                            C9
```

HOW IT WORKS

The display file in the ZX81 is not in a fixed place, but the pointer in memory locations 16396 and 16397 contain the address of the first screen location. The instruction LD DE,(16396) loads the D and E registers with the contents of 16396 and 16397—the contents of 16396 go in the E register and the contents of 16397 go in the D register, following the Z80's low-high convention. The program now knows where to start its scrolling routine.

The ZX81's screen display is also constructed rather differently from that of other machines. It has a new-line symbol at the beginning of every line (though it does not appear on the screen). These are there so that the display file can shrink to take up the minimum possible space in the ZX81's unexpanded 1K memory. If nothing is being displayed on the screen, the display file can shrink to just the 24 new-line characters, marking where each line should begin.

In the expanded versions, these new-line characters are left at the end of each line. The problem is that you must not interfere with these otherwise the computer will crash.

The first character on the screen—reading from the top left—is a new-line character. To avoid interfering with it, the instruction INC DE INCrements the DE register pair which moves the program on to the second screen location.

The two instructions LD H,D and LD L,E copy the contents of the DE register pair into the HL register pair. And INC HL increments the HL register pair to give the address of the third screen location.

LD B,24 loads the B register with the number 24. There are 24 lines on the ZX81's screen display and the B register is going to act as a counter. But the B register is going to be used for other things as well. So for the moment its contents are going to be put on to the stack with the instruction PUSH BC.

PUSH actually pushes both the B and the C register on to the stack together. But there is no instruction that pushes the B register on to the stack on its own. Anyway it doesn't much matter what is in the C register at this point as it does not affect the B register which is being used as the counter in any way.

LD A,(DE) loads the accumulator with the contents of the memory location whose address is in the DE register pair—in other words the contents of the second screen location, remember. Then LD BC,31 loads the BC registers with the number 31. There are 33 characters on a ZX81 line, but the first one, as you know, is a new-line character. And when the characters are shifted along one square to scroll them, this operation only has to be done 31 times. The other one character square has to be shifted from one end of the line to the other which is done in a separate operation.

The crucial instruction here is LDIR. This means LoaD, Increment and Repeat. What it does is load the contents of the memory location whose address is in HL—the third screen location on the first pass—into the memory location whose address is contained in DE—the second memory location on the first pass. Then it increments HL and DE, decrements BC and checks to see if BC is zero—if it is not the whole instruction is repeated.

The effect of this is to move what is in the third screen location into the second, then the fourth into the third, and so on all the way along the first row. It stops when the BC register has counted down from 31 to Ø. The last operation it does is move the contents of the 33rd screen location into the 32nd position. After that when the BC register is decremented its contents become zero and the program moves onto the next instruction.

LD (DE),A puts the contents of the accumulator into the address pointed to by the DE registers and is the complement of the instruction LD A,(DE) used earlier. That instruction was used to store the contents of the second screen location in the accumulator.

Now, the contents of the second screen location are put back into the screen location pointed to by DE. But in the meantime DE has been incremented 31 times by the LDIR instruction, so now it points to the last screen location on the line—which is exactly where we want to put the contents of the second screen location if the scrolling routine is going to wrap around.

INC HL and INC DE increment the HL and DE registers. This is done once to move the program onto the next line of the screen, and a second time to avoid interfering with the new-line character at the beginning of the line.

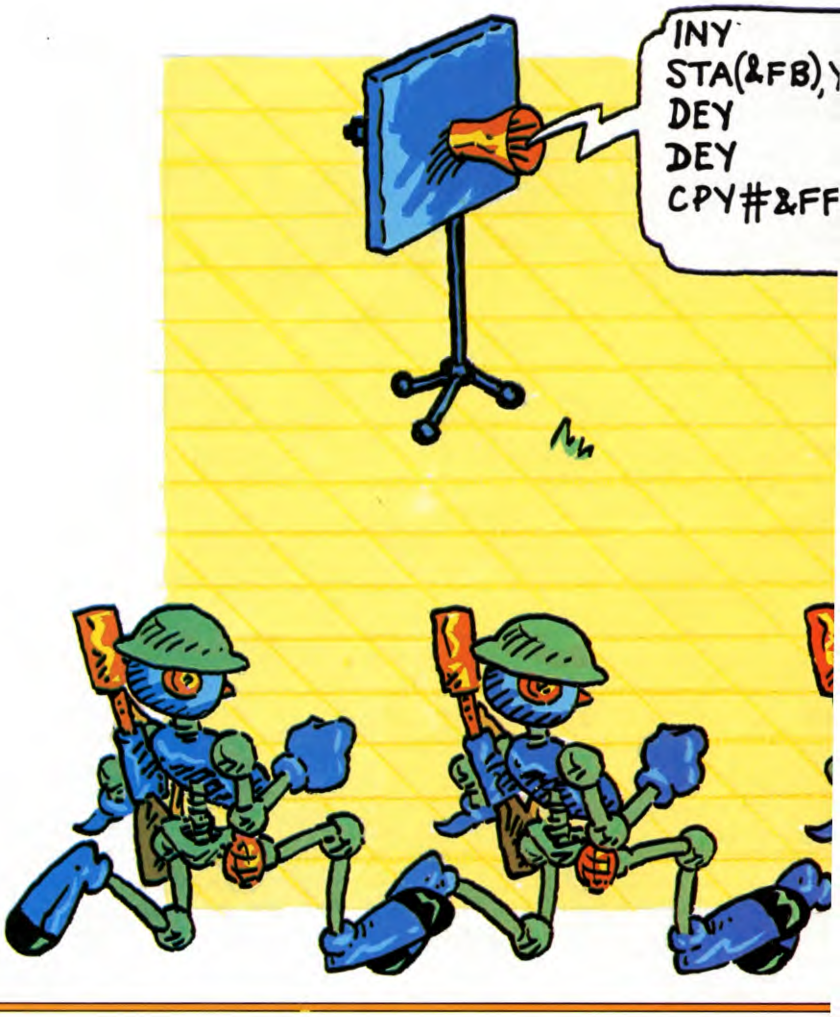
POP BC pulls the contents of the top two memory locations back off the stack and puts them into the BC registers. In other words it sets up the B register counter again.

DJNZ Decrements the B register and Jumps if its contents are Not Zero. To start with the B register was loaded with 24. This number was then stored on the stack, then pulled back again. Now it is decremented to 23. This is not zero so the instruction executes the jump—and the computer jumps back in the program to where it encounters the label LOOP again.

This jump will occur over and over again. It sends the computer back to execute the loop while the counter counts down from 24 to Ø, getting it to go through the character-shift routine a line at a time. When the contents of the B register do become Ø, the jump is not executed and the computer moves on to the next instruction.

RET means RETurn. No machine code subroutine will run safely without a return instruction, as the computer will not return to BASIC. The microprocessor will then plough on into any rubbish that follows your machine code program in memory. The chances are that the program will then crash and the memory will be corrupted.

Now try hand assembling this routine, to scroll the screen to the right, yourself. (Don't forget to convert the numbers and addresses into hexadecimal values):



```
     LD HL,(16396)
     LD DE,79Ø
     ADD HL,DE
     LD D,H
     LD E,L
     INC DE
     LD B,24
LOOP PUSH BC
     LD A,(DE)
     LD BC,31
     LDDR
     LD (DE),A
     DEC HL
     DEC HL
     DEC DE
     DEC DE
     POP BC
     DJNZ LOOP
     RET
```

This program starts at the end of the display file and works backwards through it. To do that we have to load the contents of the screen pointer into DE again, with the instruction LD DE,(16396), then add 790 to it—790 is the number of character squares there are on the ZX81's screen, minus 2. Because the screen pointer points to the first screen location, the last one is that number plus the number of screen locations minus 1. But the first character square that is going to be moved is the one before that so another 1 is subtracted.

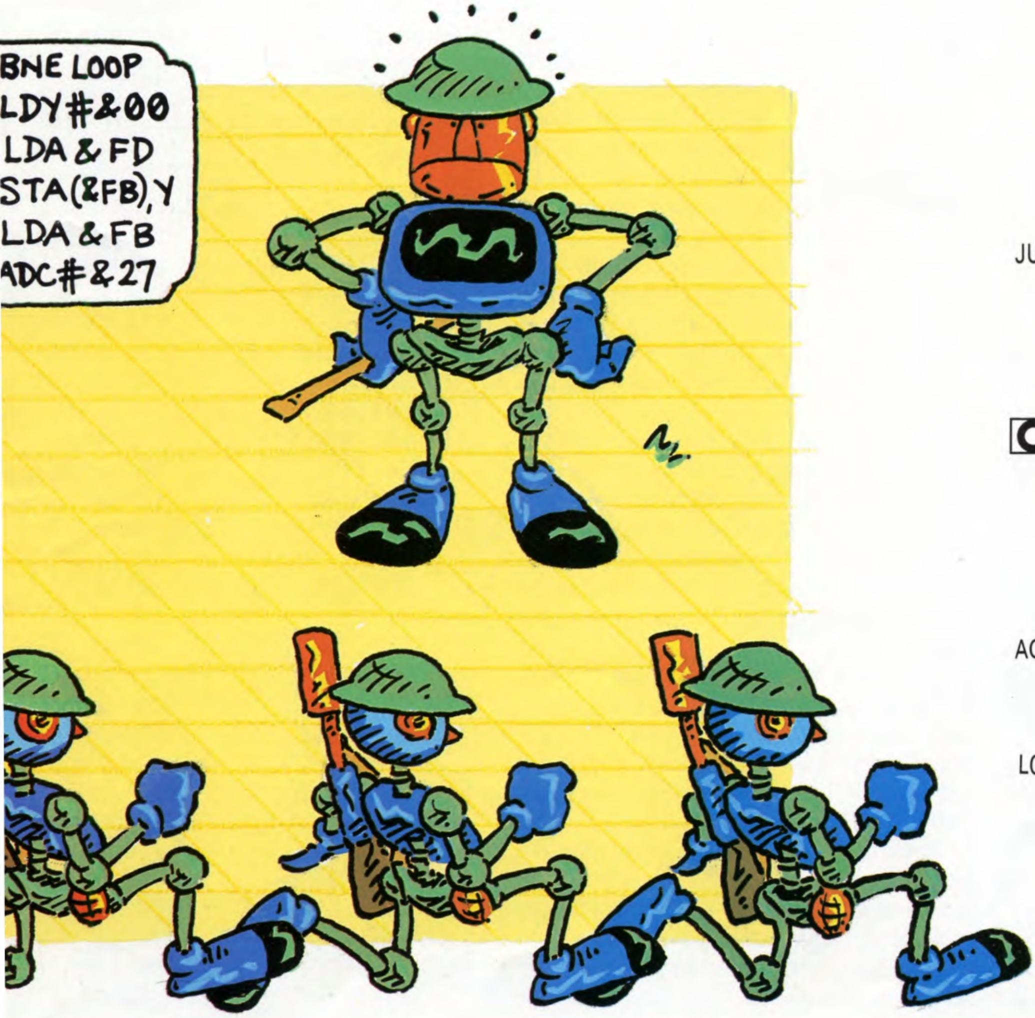
The instruction which does the addition is ADD HL,DE. This adds the contents of HL and DE and puts the result into HL. The program then goes on exactly the same as before until you get to the LDDR instruction. This is very similar to the LDIR instruction, except that LDDR LoaDs, Decrements and Repeats. In other words it loads the contents of the memory location whose address is contained in HL into the memory location whose address is in DE again, but then it decrements HL and DE, decrements BC again and repeats if the BC register does not contain zero.

The jump is the same length as before, so you can check that you have worked it out properly by referring back to the machine code listing above. (Remember to count the bytes from the end of the jump instruction.)

This assembly language routine scrolls the screen to the left. Here it has been translated into machine code for you.

```
      LDA #&ØØ       A9 ØØ
      STA &FB        85 FB
      LDA #&Ø4       A9 Ø4
      STA &FC        85 FC
      LDA #&ØØ       A9 ØØ
      STA &FE        85 FE
AGAIN LDY #&ØØ       AØ ØØ
      LDA (&FB),Y    B1 FB
      STA &FD        85 FD
      LDY #&Ø1       AØ Ø1
 LOOP LDA (&FB),Y    B1 FB
      DEY               88
      STA (&FB),Y    91 FB
      INY               C8
      INY               C8
      CPY #&28       CØ 28
      BNE LOOP       DØ F5
      LDY #&27       AØ 27
      LDA &FD        A5 FD
      STA (&FB),Y    91 FB
      LDA &FB        A5 FB
      ADC #&27       69 27
      STA &FB        85 FB
      BCC JUMP       9Ø Ø2
      INC &FC        E6 FC
 JUMP INC &FE        E6 FE
      LDX &FE        A6 FE
      CPX #&19       EØ 19
      BNE AGAIN      DØ D5
      RTS               6Ø
```

```
      LDA #&ØØ       A9 ØØ
      STA &FB        85 FB
      LDA #&1E       A9 1E
      STA &FC        85 FC
      LDA #&ØØ       A9 ØØ
      STA &FE        85 FE
AGAIN LDY #&ØØ       AØ ØØ
      LDA (&FB),Y    B1 FB
      STA &FD        85 FD
      LDY #&Ø1       AØ Ø1
 LOOP LDA (&FB),Y    B1 FB
      DEY               88
      STA (&FB),Y    91 FB
      INY               C8
      INY               C8
      CPY #&16       CØ 16
      BNE LOOP       DØ F5
      LDY #&15       AØ 15
      LDA &FD        A5 FD
```



```
         STA (&FB),Y      91 FB
         LDA &FB          A5 FB
         ADC #&15         69 15
         STA &FB          85 FB
         BCC JUMP         90 Ø2
         INC &FC          E6 FC
    JUMP INC &FE          E6 FE
         LDX &FE          A6 FE
         CPX #&17         EØ 17
         BNE AGAIN        DØ D5
         RTS              6Ø
```

HOW IT WORKS

LDA# &ØØ loads Ø into the accumulator and STA &FB stores it at memory location ØØFB. Similarly, LDA# &Ø4 and STA &FC store Ø4 on the 64 and LDA# &1E and STA &FC store 1E on the Vic in memory location ØØFC, via the accumulator. There is no command to store data directly into a memory location.

Ø4ØØ is the address of the first screen location on the 64 and 1E is the address of the first screen location on the Vic. ØØFB and ØØFC are two locations on the part of the user's workspace on the zero page. The zero page is used as locations there only require a one byte address.

Again, LDA# &ØØ and STA &FE load Ø into memory location ØØFE. This is going to be used as a counter.

LDY# &ØØ loads the index register with the first offset—Ø naturally, as the scrolling program is to begin at the start of the screen. The LDA (&FB),Y loads the accumulator with the contents of the memory location given by ØØFB, and the next byte ØØFC, plus an offset given by the contents of the Y register. ØØFB and ØØFC point to Ø4ØØ on the 64 and 1EØØ on the Vic, the start of the screen and the offset is Ø, so this instruction feeds the contents of the first character square into the accumulator. STA &FD then stores it in ØØFD.

Next, LDY# &Ø1 loads the Y register with 1. LDA (&FB),Y then loads the accumulator with the contents of the memory location whose address is given by ØØFB and ØØFC plus an offset given by the contents of the Y register again. But this time, the Y register contains 1 instead of Ø. So this instruction loads the accumulator with the contents of Ø4Ø1 on the 64 and 1EØ1 on the Vic, which is the second screen location.

DEY Decrements the Y register and STA (&FB),Y stores the contents of the accumulator into the address given by ØØFB, ØØFC and the offset in Y. As Y has been decremented by 1 in the process, this has the effect of shifting the contents of each memory location along one.

The Y register is then INcremented twice by the instruction INY, preparing it to point to the next screen location. CPY# &28 ComPares the contents of the Y register with 28 hex, or 40 in decimal on the 64. There are 40 columns on the Commodore 64's screen. CPY# &28 sets the zero flag if the contents of the Y register is 40 decimal.

CPY # &16 compares the contents of the Y register with 16 hex, or 22 decimal, on the Vic. There are 22 columns on the Vic's screen. CPY # &16 sets the zero flag if the contents of the Y register is 22 decimal.

BNE LOOP checks the zero flag. If it is not set, the microprocessor branches back to where the LOOP label occurred before and starts that section again.

BNE is the Branch if Not Equal instruction, and it will continue to send the microprocessor back round the loop as the Y register counts up from 1 to 40 on the 64, from 1 to 22 on the Vic, and moves each of the character squares along one row. When the contents of the Y register reaches 4Ø on the Commodore 64, or when it reaches 22 on the Vic, the BNE condition is not fulfilled and the microprocessor goes onto the next instruction.

LDY# &27 loads the Y register with 27 hex or 39 decimal on the 64 and LDY # &15 loads the Y register with 21 on the Vic. LDA &FD loads the accumulator with the contents of ØØFD. If you look back, you will see that ØØFD contains the first character square.

STA (&FB),Y loads the contents of the accumulator into the memory locations given by ØØFB and ØØFC—Ø4ØØ on the 64, 1EØØ on the Vic—and the offset in the Y register—27 hex on the 64, 15 hex on the Vic. So this instruction puts the contents of the first character square into memory location Ø427 on the 64 or 1E15 on the Vic, which is the last character square in the first row, thereby swapping the first character square round into the last.

LDA &FB loads the accumulator with the contents of memory location ØØFB, then ADC# &27 adds 39 to them on the 64 or ADC# &15 adds 21 to them on the Vic. The result is put back into memory location ØØFB by STA &FB. This has the effect of setting the program up to handle the next row of the screen.

BCC means Branch on Carry Clear and ADC means ADd with Carry. So if the ADC instruction does not overflow the eight-bit accumulator—and thus the carry flag is not set—the BCC makes the microprocessor jump onto the INC &FE instruction.

But if the ADC operation does overflow the accumulator and sets the carry flag, the microprocessor moves on to INC &FC. This simply increments the contents of memory location ØØFC which contains the high byte of the screen location pointer. This ensures that the pointer is incremented properly and nothing is lost in the works.

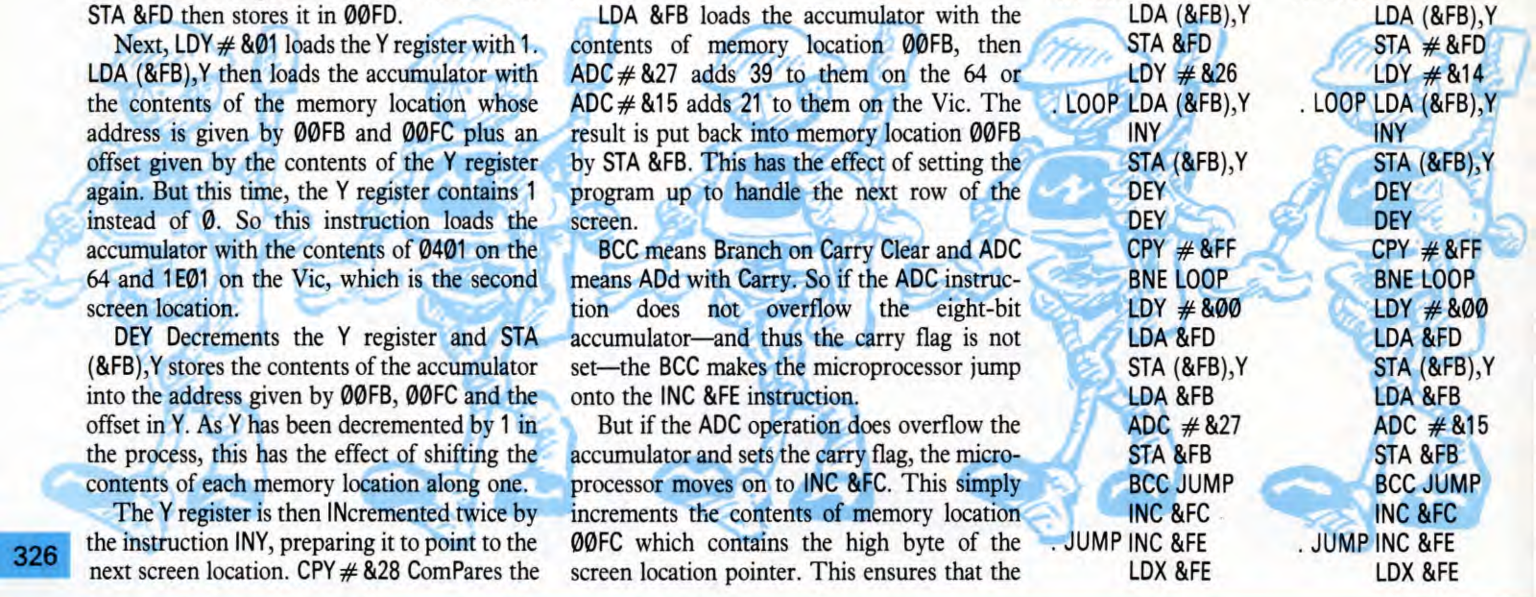
INC &FE increments the counter in ØØFE. And LDX &FE loads the contents of ØØFE into the X register so that CPX# &19 can compare it with 25—the number of rows on the Commodore 64's screen or CPX# &17 can compare it with 23, the number of rows on the Vic 20's screen.

If the counter in ØØFE is less than 25 on the 64 or 23 on the Vic, the CPX does not set the zero flag, so the BNE instruction operates and sends the microprocessor back to the beginning of the routine to move the next row along one. But if the ØØFE counter has clocked up 25 on the 64 or 23 on the Vic, the BNE condition is not fulfilled and the microprocessor goes on to the next instruction.

RTS tells the microprocessor to return to BASIC. All machine code subroutines must end with RTS, otherwise the microprocessor will career on up memory, trying to perform any piece of garbage it might find there and crash. If you're lucky, you may be able to save your program by RUN/STOP RESTORE.

Note that the program only scrolls characters, not the colours. So you might have to fill in the screen with a colour different from the background colour before using the scroll.

The following assembly language program scrolls the screen to the right. Try hand assembling it yourself.



```
        LDA #&ØØ
        STA &FB
        LDA #&Ø4
        STA &FC
        LDA #&ØØ
        STA &FE
. AGAIN LDY #&27
        LDA (&FB),Y
        STA &FD
        LDY #&26
. LOOP  LDA (&FB),Y
        INY
        STA (&FB),Y
        DEY
        DEY
        CPY #&FF
        BNE LOOP
        LDY #&ØØ
        LDA &FD
        STA (&FB),Y
        LDA &FB
        ADC #&27
        STA &FB
        BCC JUMP
        INC &FC
. JUMP  INC &FE
        LDX &FE
```

```
        LDA #&ØØ
        STA &FB
        LDA #&1E
        STA &FC
        LDA #&ØØ
        STA &FE
. AGAIN LDY #&15
        LDA (&FB),Y
        STA #&FD
        LDY #&14
. LOOP  LDA (&FB),Y
        INY
        STA (&FB),Y
        DEY
        DEY
        CPY #&FF
        BNE LOOP
        LDY #&ØØ
        LDA &FD
        STA (&FB),Y
        LDA &FB
        ADC #&15
        STA &FB
        BCC JUMP
        INC &FC
. JUMP  INC &FE
        LDX &FE
```

```
    CPX #&19                  CPX #&16
    BNE AGAIN                 BNE AGAIN
    RTS                       RTS
```

This program, you will notice, works the same as the scroll left program—except in the line shift routine you increment where you decremented before, and decrement where you incremented.

The branches are the same, so you will be able to check the jumps by looking back at the machine code listing above. (Remember that you have to work out the jump from the end of the branch instruction, that is including the byte that carries the jump itself.)

As the Electron and the BBC Micro use essentially the same chip as the Commodores, the same scrolling programs should work on both—with adjustments as the memory addresses are not the same as the Commodore's. The MODE 7 screen starts at 7C00 instead of 0400, so the 04 in the third line has to be changed to 7C. And all the zero page addresses have to be changed as the BBC Micro uses different parts of the zero page as its own workspace.

The zero page addresses FB, FC, FD and FE in the Commodore's programs must be changed to 70, 71, 72 and 73 for the BBC. This is covered in the user guide and in a later article in *INPUT*.

Unfortunately these programs only work properly on the first 25 lines in Mode 7 so you cannot use them on the Electron.

In fact, this whole machine code routine is unnecessary on the BBC Micro, since the screen instruction VDU 23;13,V,0;0;0; (where V runs from 0 to 40) scrolls the screen to the left and (where V runs in the other direction from 40 to 0) scrolls it to the right. This does not work on the Electron either.

This assembly language routine scrolls the screen to the left. Here it has been translated into machine code for you.

```
     LDX #1024              8E 00 00
LOOP LDB ,X+                   E6 80
     PSHS B                    34 04
     LDB #31                   C6 1F
JUMP LDA ,X+                   A6 80
     STA -2,X                  A7 1E
     DECB                         5A
     BNE JUMP                  26 F9
     PULS B                    35 04
     STB -1,X                  E7 1F
     CMPX #1536             8C 06 00
     BLO LOOP                  25 EA
     RTS                          39
```

HOW IT WORKS

The computer's text screen begins at 1024, so LDX#1024 loads the address of the top left character square into the X register. LDB ,X+ loads the B accumulator with the contents of the memory location whose address is in the X register and increments the X register by 1. PSHS B PuSHes the contents of the B accumulator—that is the contents of the first character square—on to the stack S.

LDB#31 loads B with the number 31. This is going to be used as a counter. Although there are 32 character squares in a line on the Dragon and Tandy you only do the simple operation of shifting each of them one square to the left 31 times. Taking the contents of the first square and shifting it to the other end is handled separately after the rest of the line has been shifted.

LDA ,X+ loads the A accumulator with the contents of the memory location whose address is contained in the X register—this you will note is one on from when we loaded the B accumulator, because the X register was incremented afterwards—and increments the X register again. The STA −2,X then STores the contents of the A accumulator in the memory location two before the one now pointed to by the X register, in other words one before the one it was taken from—the X register has been incremented since then, remember.

DECB DECrements the B register, counting it down on each successive pass from 31 to 0. This operation affects the zero flag—so if the result is zero the zero flag is set. BNE checks to see if the zero flag is set. If it is not the program branches. JUMP is the label so the program loops back to the LDA ,X+ instruction and shifts the contents of the next character square along until it gets to the end of the line.

At the end of the line—when DECB has counted the B register down to 0—the zero flag is set, so the branch does not occur and the microprocessor goes on to the next instruction in the program.

PULS B PULls the contents of the top memory location off the stacks and puts it back into the B register. The contents of the B register are then stored in the memory location whose address is one less than the one in the X register. This is the last character square in the row and what you got back off the stack, you remember, is the contents of the first character square. So the whole row has been shifted along and the first square in the row has been shifted round to the end.

CMPX#1536 CoMPares the address in the X register with 1536, which is the address of the first memory location after the end of the text screen. And BLO branches if contents of the X register is lower than 1536. LOOP is a label so if the program has not completed shifting the last character square the microprocessor jumps back to the LDB ,X+ instruction and starts again on the next line of the text screen.

If the X register does contain 1536, the last character square on that line has been shifted, and the microprocessor goes onto the next instruction.

RTS returns to BASIC. Every machine code subroutine must end with this instruction, otherwise the microprocessor will carry on up the memory trying to perform any instruction it might find there and end up crashing.

The following assembly language program scrolls the screen to the right. Try hand assembling it yourself. (Remember, the numbers and addresses given are in decimal, don't forget to convert them into hex.)

```
     LDX #1536
LOOP LDB ,-X
     PSHS B
     LDB #31
JUMP LDA ,-X
     STA 1,X
     DECB
     BNE JUMP
     PULS B
     STB ,X
     CMPX #1024
     BGT LOOP
     RTS
```

This program, you will notice, works the other way round. It starts at the end of the screen in location 1536 and works back to the beginning at 1024. Actually 1536 is the location after the end of the screen but the postbyte ,−X decrements the X register before the command is executed, as opposed to ,X+ which increments the X register after the instruction has been executed. So if LDB ,−X is going to load the B accumulator with the contents of the last screen location, you have to start with the address of the one after the last screen location in the X register.

Otherwise, this program works in exactly the same way as the previous one, so you will be able to check that you have worked out the branches right by looking back to the machine code listing given above.

DON'T PANIC

If you have followed this article you already know that assembling by hand is extremely difficult. But don't panic. In the next few issues, *INPUT* is publishing assemblers which will work on the Spectrum, Commodore 64, Dragon and Tandy computers and will do all this tedious translation for you. The Acorn's already have a built-in assembler.

A COMPUTER TYPING TUTOR-2

Now that you are proficient at typing the middle line of keys you can extend your skills to cover the whole alphabet. But first you'll have to alter your existing program . . .

Last time you were given the first part of *INPUT*'s typing course. Once you've mastered the middle line of the keyboard to a reasonable standard—say 15 words per minute without making any mistakes—you can progress to the second stage.

THE QWERTY KEYS

By adding these lines to your existing program you can add the top line of letters—the 'QWERTY' line from which the keyboard layout often takes its name—to your repertoire. LOAD your program as it stands, then type in these new lines. Some of them overwrite lines which are no longer needed—others are completely new.

```
30 LET S$="QAWSEDRFTGYHUJIKOLP"
210 FOR K=6 TO 24
230 LET R$=S$(K-5)
320 LET RN=INT (RND*19)+1
330 PRINT AT 10,RN+5;"*": LET
   R$=S$(RN)
350 PRINT AT 10,RN+5;"□"
440 LET RN=INT (RND*19)+1
530 PRINT AT 10,13;"□□□□□
   □□□□□": PRINT AT 10,13;T$
540 FOR M=1 TO LEN T$: PRINT AT
   9,11+M;"□*□□□□□□□
   □□□"
610 FOR N=1 TO 4: RESTORE : LET
   RN=INT (RND*24)+1: FOR K=1 TO RN:
   READ X$: NEXT K
1010 PRINT AT 12,6;S$
2000 DATA "TWIST","QUEST",
   "TRADE","POISED","GRAPE",
   "PRIEST","THRASH","WHALE",
   "ADDRESSED","TRAPPED",
   "SLOWLY","PLAGUE"
2010 DATA "TULIP","PLEASE",
   "RUDDER","JESTER","OTHER",
   "FLOUR","YOUTH","AFTERWARDS",
   "QUARTER","FRAGILE","YEAST",
   "JAIL"
```

```
10 DIM W$(21),WO$(28):FOR Z=1TO28:
   READ WO$(Z):NEXT
40 A$="QAWSEDRFTGYHUJIKOLP:":
   POKE 54296,15:GOTO 380
80 PRINT"▤▣▣▣▣▣";:
   IF K<3 THEN PRINTTAB(11)
   "◣▣"A$:POKE 198,0
90 X=INT(RND(1)*20)+1:N=N+1:
   P=P+1:IF K=1 THEN X=N:
   GOTO 120
100 IFK=3THENX=INT(RND(1)*20)
   +1:PRINTTAB(18)"◣◺▤◸▣
   ▮▮▮▮▮▮◫▣"MID$(A$,X,1)
   "◣◫▣▮▮▮▮▮▮◹▤◿"
```

- EXTENDING THE TYPING TUTOR
- ADDING THE TOP LINE OF KEYS TO THE PROGRAM
- HOW TO POSITION YOUR FINGERS FOR THE EXTRA KEYS
- ADDING THE BOTTOM LINE OF KEYS
- CHANGING THE DATA
- PRACTISING WITH THE WHOLE ALPHABET

```
110 IF K=4 THEN PRINT TAB(16)
   "◣□□□□□□□□□□
   ▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌"
   W$(WW):X=N+5
120 IF K<3 OR K=4 THEN PRINT
   "▤▣▣▣▣"TAB(11+(X−1))
   "▟*"
130 IF K=5 THEN FOR Z=1 TO 5:
   PRINT "π"W$(Z);:NEXT Z:
   PRINT:PRINT "▚◣";
220 IF K<3 OR K=4 THEN PRINT
   "▤▣▣▣▣"TAB(11+(X−1))"□"
260 IF K=1 AND N=20 THEN 290
350 NU=0:P=0:S=0:FORWW=1TOMM:
   W$(WW)=WO$(INT(RND(1)*28)+1)
360 IF K=5 AND WW<>5 THEN
   W$(WW)=W$(WW)+"□"
470 IF K=5 THEN PRINT TAB(12)
   "◘TYPE THESE WORDS":MM=5
540 DATA TWIST,QUEST,TRADE,
   POISED,GRAPE,PRIEST,THRASH,
   WHALE,ADDRESSED
550 DATA TRAPPED,SLOWLY,PLAGUE,
   TULIP,PLEASE,RUDDER,JESTER,
   OTHER,FLOUR
560 DATA YOUTH,AFTER,QUARTER,
   FRAGILE,YEAST,JAIL,WEATHER,
   TOWARDS,REWARD,THROUGH
```

```
530 PRINTTAB(−1+T*2,9)"□"
   TAB(−1+T*2,10)B2$
630 PRINTTAB(1,10)A2$
650 P=RND(20):X=P*2−1
820 P=RND(20):B2$=MID$(A$,P,1)
950 P=RND(24)
1150 B2$=A$(RND(24))
1180 B2$=B2$+"□"+A$(RND(24))
1330 DATA TWIST,QUEST,TRADE,
   POISED,GRAPE,PRIEST,THRASH,
   WHALE,DRESSED,TRAPPED,
   SLOWLY,PLAGUE,JAIL
1340 DATA TULIP,PLEASE,RUDDER,
   JESTER,OTHER,FLOUR,YOUTH,
   AFTER,QUARTER,FRAGILE,
   YEAST
```

```
20 DIM A$(24)
30 FOR T=1 TO 24: READ A$(T): NEXT
40 A$="QAWSEDRFTGYHUJIKOLP;":
   A2$="Q□A□W□S□E□
   D□R□F□T□G□Y□H□U□
   J□I□K□O□L□P□;"
420 PRINTTAB(1,10)A2$
430 FOR T=1 TO 20
460 PRINTTAB(−1+T*2,9)"*"
   TAB(−1+T*2,10)B2$
```

```
10 OB$="QAWSEDRFTGYHUJIKOLP;"
210 AP=1253
220 FOR K=1 TO 20
230 AP=AP+1
320 AP=1253+RND(20)
430 P$=MID$(OB$,RND(20),1)
800 CLS:P$="":FOR K=1 TO 4
9000 DATA TWIST,QUEST,TRADE,
   POISED,GRAPE,PRIEST,THRASH,
   WHALE,ADDRESSED,TRAPPED,
   SLOWLY,PLAGUE
9010 DATA TULIP,PLEASE,RUDDER,
   JESTER,OTHER,FLOUR,YOUTH,
    AFTER,QUARTER,FRAGILE,YEAST,
    JAIL
9Ø2Ø DATA WEATHER,TOWARDS,
    REWARD,THROUGH
```

When you RUN the program, the screen will display the familiar menu of five options. Level 1 displays the sequence of letters QAWSEDRFTGYHUJIKOLP. Depending upon which computer you have, the layout of the keyboard may also include a final character ; or :. As a prompt, you will see the asterisk appearing above them, working from left to right as before.

Levels 2 and 3 work in exactly the same way as in the first test, getting you to type these characters at random but with a greater range of letters.

THE ZXCV KEYS

When you've become proficient at using the top line of keys with the middle line it's time to try the bottom line.

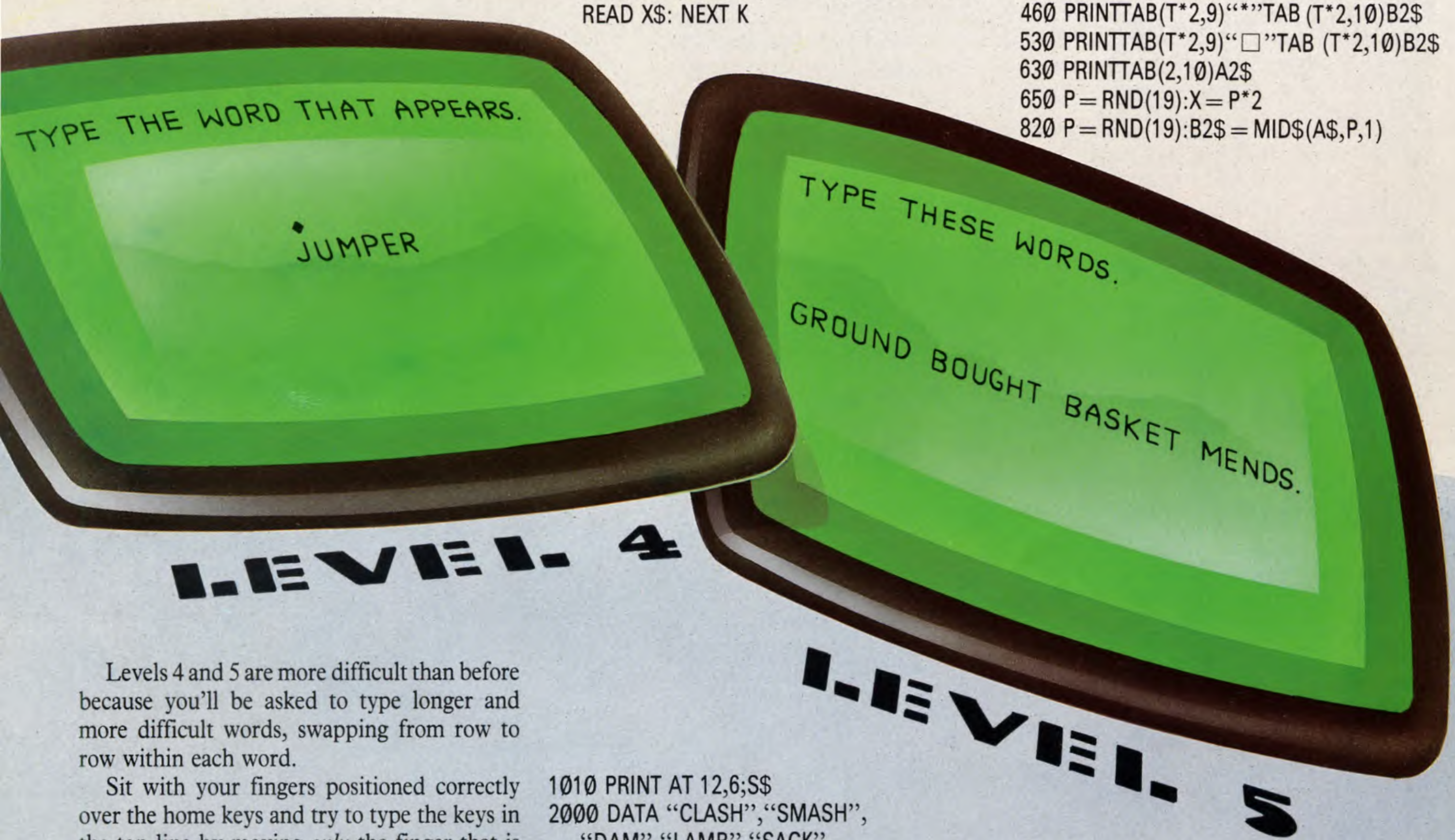
Part three of the course teaches you how to use the bottom and middle lines of the keyboard, progressing through each of the five levels of skill—but not yet all three together.

Here are the alterations you'll have to make this time:

```
3Ø LET S$ = "AZSXDCFVGBHNJMKL"
21Ø FOR K = 6 TO 21
32Ø LET RN = INT (RND*16) + 1
44Ø LET RN = INT (RND*16) + 1
61Ø FOR N = 1 TO 5: RESTORE : LET
    RN = INT (RND*24) + 1: FOR K = 1 TO RN:
    READ X$: NEXT K
```

```
54Ø DATA CLASH,SMASH,
    DAM,LAMB,SACK,
    SLACK,LAX,SHAM,
    HAMS,SLAM
55Ø DATA MADAM,FLAN,
    JAZZ,KNACK,CLASS,
    VAN,LAVA
56Ø DATA SHALL,BAND,
    BLACK,CASH,JACK,
    SLANG,CALL,BALL,
    HAND,GNASH,CASK
```

```
4Ø A$ = "AZSXDCFVGBHNJMK,L.;":
    A2$ = "A□Z□S□X□
    D□C□F□V□G□B□H□
    N□J□M□K□,□L□.□;"
42Ø PRINTTAB(2,1Ø)A2$
43Ø FOR T = 1 TO 19
46Ø PRINTTAB(T*2,9)"*"TAB (T*2,1Ø)B2$
53Ø PRINTTAB(T*2,9)"□"TAB (T*2,1Ø)B2$
63Ø PRINTTAB(2,1Ø)A2$
65Ø P = RND(19):X = P*2
82Ø P = RND(19):B2$ = MID$(A$,P,1)
```



Levels 4 and 5 are more difficult than before because you'll be asked to type longer and more difficult words, swapping from row to row within each word.

Sit with your fingers positioned correctly over the home keys and try to type the keys in the top line by moving *only* the finger that is needed for each one, not your whole hand. This time use the little finger of your left hand for Q as well as A, the next finger for S and W, and so on, as far as the little finger on your right hand which is used for; (or whatever, if you have it) and P. Your index fingers will have to work harder again—the index finger of your left hand will be used for F, G, R and T, whilst the other index finger will be used for H, J, Y and U. After you're pressed a key on the top line your finger should return to its position over the right home key.

```
1Ø1Ø PRINT AT 12,6;S$
2ØØØ DATA "CLASH","SMASH",
    "DAM","LAMB","SACK",
    "SLACK","LAX","SHAM",
    "HAMS","MADAM","JAZZ",
    "KNACK"
2Ø1Ø DATA "CLASS","VAN",
    "LAVA","SHALL","BAND",
    "BANG","BLACK","CASH",
    "JACK","CALL","BALL",
    "GNASH"
```

```
4Ø A$ = "AZSXDCFVGBHNJMK,L.:/":
    POKE 54296,15:GOTO 38Ø
```

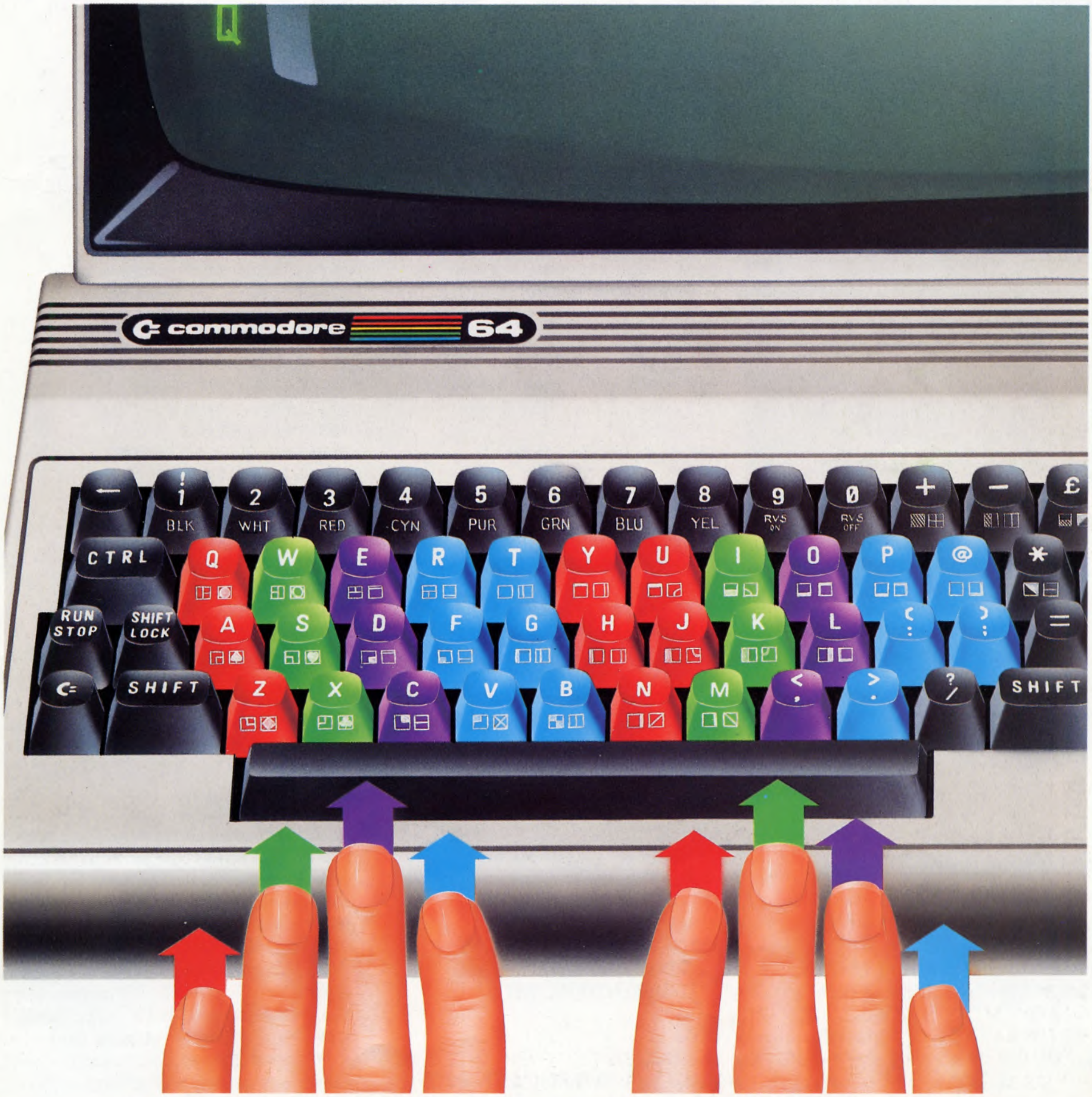
```
133Ø DATA CLASH,SMASH,DAM,LAMB,
    SACK,SLACK,LAX,SHAM,HAMS,
    MADAM,JAZZ,KNACK
134Ø DATA CLASS,VAN,LAVA,SHALL,
    BAND,BANG,BLACK,CASH,JACK,
    CALL,BALL,GNASH
```

```
1Ø OB$ = "AZSXDCFVGBHNJMK,L.;"
22Ø FOR K = 1 TO 19
32Ø AP = 1253 + RND(19)
```

```
430 P$=MID$(OB$,RND(19),1)
800 CLS:P$="":FOR K=1 TO 5
9000 DATA CLASH,SMASH,
   DAM,LAMB,SACK,SLACK,LAX,
   SHAM,HAMS,SLAM,MADAM,FLAN,
   JAZZ,KNACK
9010 DATA CLASS,VAN,LAVA,
   SHALL,BAND,BANG,BLACK,
   CASH,JACK,SLANG
9020 DATA CALL,BALL,GNASH,
   CASK
```

RUNning the program this time will allow you to progress through the five lessons, but with letters and words drawn from the bottom two rows of the keyboard.

Again, sit with your fingers correctly positioned on the home keys. This time you'll be moving your fingers down to the bottom line before returning to the correct home key. The little finger of your left hand will operate the A and Z, the next finger the S and X, and so on. The index finger of your left hand will be used for the F, G, V and B, whilst the index finger of your right hand will be used for H, J, N and M. The remaining fingers have to operate the punctuation keys in the bottom line, which vary from keyboard to keyboard. On the Spectrum, these are not available without using SYMBOL SHIFT and will be covered later. Note that the punctuation keys are not included in the word tests—you'll get a chance to practise punctuation with a further program in a later article.


Q
commodore 64
← 1 BLK 2 WHT 3 RED 4 CYN 5 PUR 6 GRN 7 BLU 8 YEL 9 RVS ON 0 RVS OFF + − £
CTRL Q W E R T Y U I O P @ *
RUN STOP SHIFT LOCK A S D F G H J K L : ; =
C= SHIFT Z X C V B N M < , > . ? / SHIFT

Microtip

Improving the Spectrum's keyboard

You can improve upon the Spectrum's keyboard and so make it easier to learn to type, by replacing the Sinclair standard keyboard with a new one.

Several companies now sell 'real' keyboards for the Spectrum and although there are a number of differences they are all very easy to fit to your machine.

The easiest to fit comes as a totally separate unit that simply plugs into the back of the computer. The old keyboard is then disabled and you do all your typing on the new keyboard.

Other types replace the original keyboard and with these you have to open up the computer, disconnect the two ribbon cables from the old keyboard and connect them to the new one.

The problem with this method is that your guarantee becomes invalid if you open up the Spectrum's case. However you can overcome this by taking out an insurance policy against your Spectrum going wrong.

THE WHOLE ALPHABET

Once your proficiency at using the bottom row of keys matches your proficiency at the top row, you can continue with the next stage of the course. This time you can practise using the whole alphabet.

Here are the alterations you need to make to your existing program:

```
30 LET S$="QAZWSXEDCRFVTGBYHN
   UJMIKOLP"
210 FOR K=2 TO 27
230 LET R$=S$(K-1)
320 LET RN=INT (RND*26)+1
330 PRINT AT 10,RN+1;"*": LET
   R$=S$(RN)
350 PRINT AT 10,RN+1;"□"
440 LET RN=INT (RND*26)+1
530 PRINT AT 10,13;"□□□□□
   □□□□□":PRINT AT 10,13;T$
540 FOR M=1 TO LEN T$: PRINT AT 9,11
   +M;"□*□□□□□□□□□"
610 FOR N=1 TO 5: RESTORE : LET
   RN=INT (RND*24)+1: FOR K=1 TO RN:
   READ X$: NEXT K
1010 PRINT AT 12,2;S$
2000 DATA "QUIZ","THROUGH",
   "WIND","MURDER","JUNTA",
   "FROWN","BUNK","WAXY",
   "WHACK","KILN","SWARM",
   "LONGER"
2010 DATA "TICKLE","LIMBS",
   "VIEWED","SETTING","PACK",
   "CRAVE","DRIBBLE","MEANT",
   "MAWLED","WRIGGLE","ANSWER",
   "WINDOW"
```

As with all the programs in this typing course, as soon as you become familiar with the words in the DATA statements you should change them for some new ones. Often the programs for the other computers have different words that you can use, but make sure that you don't use fewer words than the original program or you'll be given an OUT OF DATA error. You can have more entries, of course, but these won't be READ by the computer unless you change the program.

```
40 A$="QAZWSXEDCRFVTGBYHNUJMIK,
   OL.P:/":POKE 54296,15:GOTO 380
80 PRINT "▤▣▣▣▣▣";:
   IF K<3 THEN PRINT TAB(5)
   "◣▣"A$:POKE 198,0
90 X=INT(RND(1)*30)+1:N=N+1:
   P=P+1:IF K=1 THEN X=N:
   GOTO 120
100 IFK=3THENX=INT(RND(1)
   *30)+1:PRINTTAB(18)
   "◣◿⊟◺▣▮▮▮▮▮▮ | ▣"
   MID$(A$,X,1)"◣ | ▣▮▮
   ▮▮▮◹⊟◸ "
110 IF K=4 THEN PRINT TAB(16)
   "◣□□□□□□□□□□
   ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮"
   W$(WW):X=N+11
120 IF K<3 OR K=4 THEN PRINT
   "▤▣▣▣▣"TAB(5+(X-1))
   "▣*"
220 IF K<3 OR K=4 THEN PRINT"▤
   ▣▣▣▣"TAB(5+(X-1))"□"
260 IF K=1 AND N<30 THEN 70
280 IF (K<4 AND K>1) AND N<20 THEN
   70
540 DATA TYPING,THROUGH,THINK,
   VIEWED,SETTING,PACK,CRAVE,
   MEANT,LIMBS,TICKLE
550 DATA LONGER,SWARM,WANT,
   MATE,MURDER,WIND,QUIZ
560 DATA WINDOW,CAR,NOTHING,
   CARAVAN,NAME,SLAVE,BORDER,
   ZERO,BREAD,MODE,SAVE
```

```
20 DIM A$(29)
30 FOR T=1 TO 29:READ A$(T): NEXT
40 A$="QAZWSXEDCRFVTGBYHNUJ
   MIK,OL.P;":A2$=A$
420 PRINTTAB(4,10)A2$
430 FOR T=1 TO 29
460 PRINTTAB(3+T,9)"*"TAB (3+T,10)B2$
530 PRINTTAB(3+T,9)"□"TAB
   (3+T,10)B2$
630 PRINTTAB(3,10)A2$
650 P=RND(29):X=P+2
820 P=RND(29):B2$=MID$(A$,P,1)
950 P=RND(29)
1150 B2$=A$(RND(29))
1180 B2$=B2$+"□"+A$(RND(29))
1330 DATA QUIZ,THROUGH,WIND,
   MURDER,JUNTA,FROWN,BUNK,
   WAXY,WHACK,KILN,SWARM,
   LONGER,TICKLE,LIMBS
1340 DATA VIEWED,SETTING,PACK,
   CRAVE,DRIBBLE,MEANT,SKULL,
   CASTLE,WANDER,DUCK,
   DECIMAL,VAMPIRE,WOMEN,
   ACORNS,SQUID
```

```
10 OB$="QAZWSXEDCRFVTGBYHNUJ
   MIK,OL.P;"
210 AP=1248
220 FOR K=1 TO 29
320 AP=1248+RND(29)
430 P$=MID$(OB$,RND(29),1)
800 CLS:P$="":FOR K=1 TO 4
1020 PRINT@257,OB$
9000 DATA QUICK,GROWN,LAZY,
   JUMPER,TRACE,VEIL,BLAZE,
   VIEWS,FAVOUR,BASKET
9010 DATA BRASH,RAMBLE,BOUGHT,
   PLANK,WANDER,MENDS,MUDDLE,
   BLANKET,GROUND
9020 DATA FREEZE,BRAKE,BARRED,
   LINKS,GRAINS,CHASED,BREAD,
   PHONE,LIMPED
```

Work your way through the five lessons as before. Remember to return your fingers to their correct position above the home keys each time after pressing a key in the top or bottom row.

GETTING BETTER

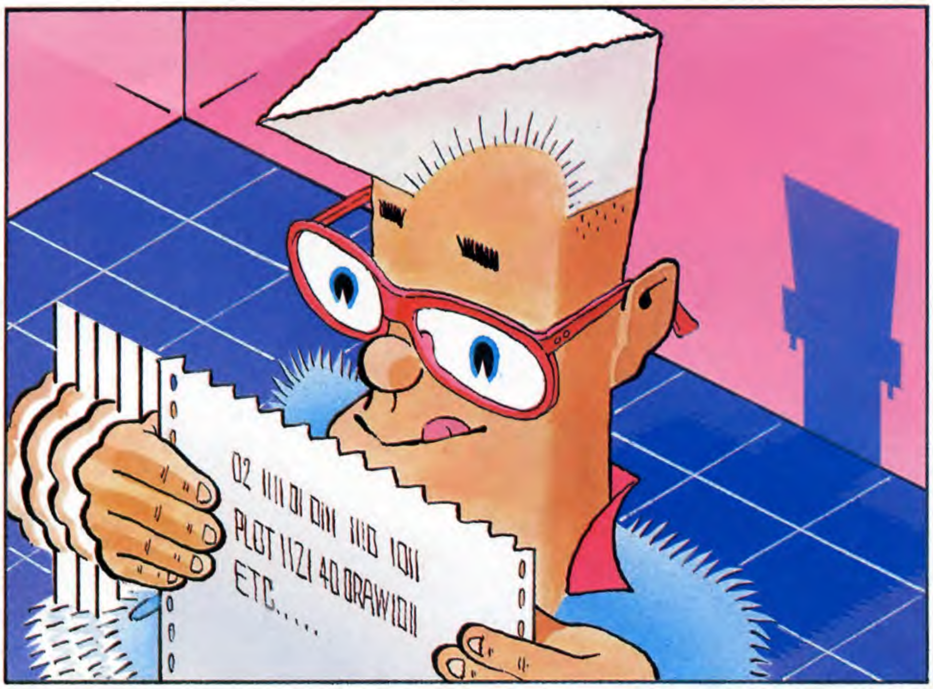
As the course progresses, you will get a chance to improve your speed and accuracy on all the letter characters on the keyboard. You'll also see how to include the number keys and punctuation—crucial for accurate copying of program listings.

By adding these and the control keys, such as SHIFT, you'll rapidly be in a position where you are completely at home typing complex sentences and lengthy program lines—and there are plenty of exercises to come which will allow you to practise your growing skills.

SHORTENING PROGRAMS

- USING ABBREVIATIONS
- HOW TO SAVE MONEY
- MISSING OUT SPACES
- USING LONG LINES
- SPEEDING UP YOUR PROGRAMS

Shorter programs mean less typing, they also take up less memory space and make your programs run faster too. Here are some tricks you can use and some pitfalls to avoid



One of the most important rules to remember when writing a program is to make is clear and readable. This usually means keeping the program lines short, adding lots of space and using long variable names where possible.

However, long readable programs do involve a lot of extra typing, as well as using up extra memory and running slightly slower than they might.

Perhaps the first reason for wanting to shorten a program is to save typing so many characters. If you are copying a listing from a magazine then there really is no reason to type in all the REM statements. After all, you can always refer back to the magazine if you forget what is going on.

Another way to reduce the wear on your typing fingers (or finger) is to miss out optional keywords. This is not possible on the Spectrum because of the structure of its BASIC—and since the keywords are entered with a single keystroke, there wouldn't be much saving anyway. But on the other computers you can miss out every LET and THEN, and if you have a line that reads IF...THEN GOTO... you can choose whether to miss out the THEN or the GOTO (but not both).

The Commodore and Acorn computers allow you to enter keywords in a shortened form, such as P. instead of PRINT on the Acorn, ? instead of PRINT on the Commodore and so on for most of the other keywords. The Dragon, though, has only two shortened forms, ? for PRINT and ' for REM.

In all but the last case, when you LIST or print out the program, the words will appear in their full form. So although it saves typing it won't save money or make the program run any faster. The next few tricks do both though.

SAVING MEMORY

Perhaps the most important reason to shorten a program is to save precious memory space. With a large program it may make the difference between one that RUNs and one that gives you 'out of memory' or 'no room' error. So here's a few ways to save some extra bytes.

One way is to use single letter names for the variables. This saves typing, saves memory and also allows the program to RUN faster. But it's always best to keep a list of what each variable does since the letters you use are unlikely to give you much of a clue when you come back to the program later on.

Another way of shortening a program is to miss out spaces. This doesn't apply to the Spectrum of course, since spaces appear automatically as you type in the program—although they don't take up aňy memory. On the other computers, missing out the spaces is really quite a desperate measure as it makes the program very difficult to read. But if you are very short on memory then it may be necessary. There is just one pitfall to be aware of—you *must* leave a space between a variable and the start of a keyword. So a line like:

```
IF A=B AND B=C THEN PRINT "OK"
```

can be shortened to

```
IFA=B ANDB=C THENPRINT"OK"
```

But if, instead, you wrote

```
IF A=BAND B=C THENPRINT"OK"
```

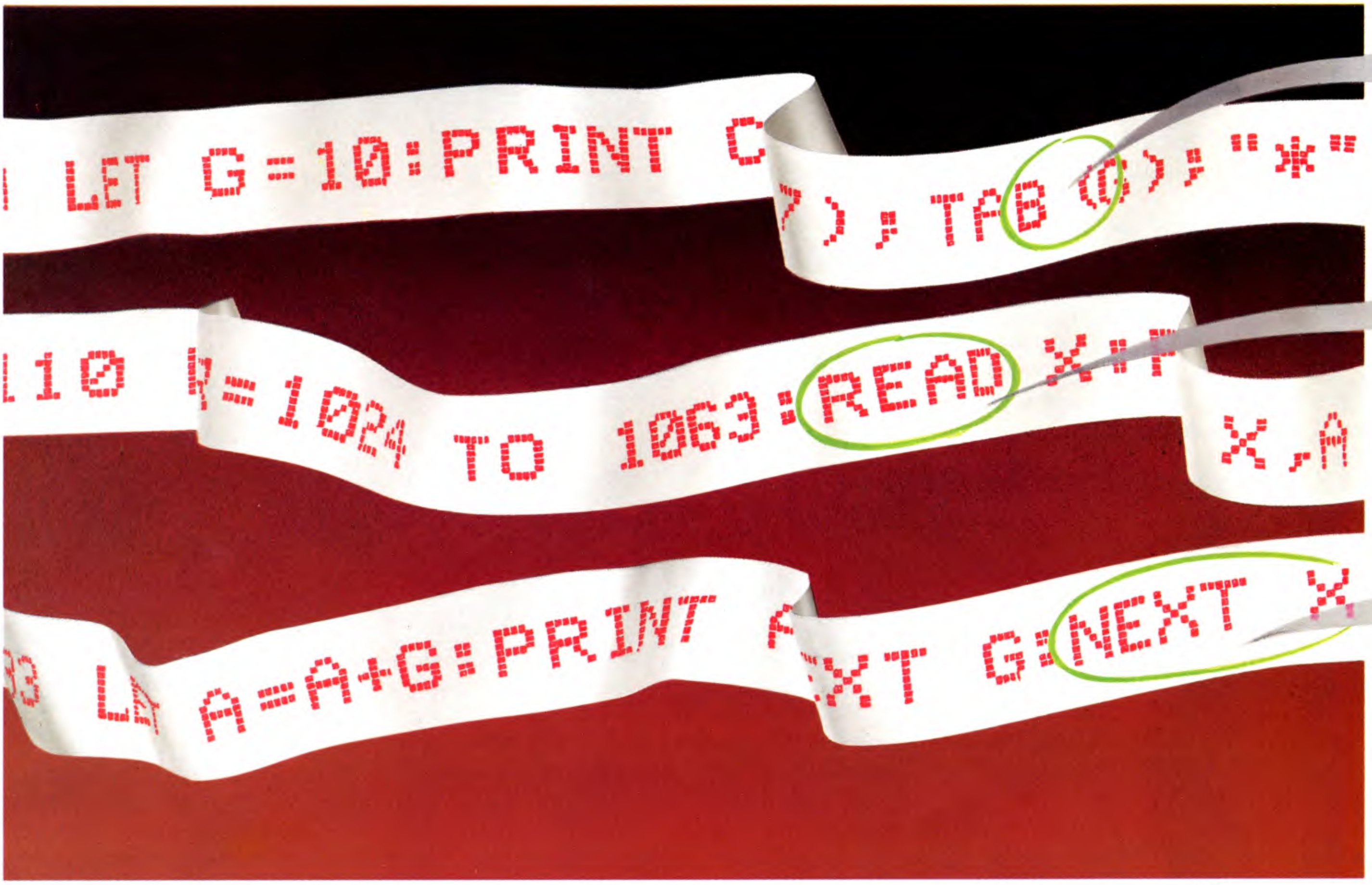
the computer would start looking for a variable called BAND, and as this doesn't exist you'd get an error message. However, you can join any number of keywords together without spaces.

The last memory saving trick is to use long program lines by combining as many statements as possible onto a single line. This saves the memory that would have been used for all the extra line numbers. But again, it makes the program almost impossible to read so it's not worth doing unless it's absolutely necessary. In any case it is not possible to do this on the ZX81.

Most of these methods tend to speed up the running of a program. But one way to add a lot of extra speed is to miss out the variable that comes after NEXT in a FOR...NEXT loop. Also if you have several loops nested together than you can put all the variables on one line as long as they are separated by commas—for example, NEXT A,B,C. And if you have an Acorn computer, you can write the last example as NEXT,,.

GETTING RID OF BUGS

Program faults are not just a nuisance, but must be located and cured before a program is of any real use. Tracing them is rather more than half the battle . . .



Program errors—'bugs'—occur in programs for a variety of reasons. No program of any length which is being developed or simply copied from a listing is likely to be completely free of them.

Program bugs can take two forms. The really serious types can prevent a program from RUNning at all. Other types may remain for the most part hidden, until a certain routine or sequence of keystrokes is brought into play. For example, in an adventure game, everything might work perfectly, unless you walk down a particular path while carrying a knife—in which case you fall down a hole that shouldn't be there. Or in an accounts program you might suddenly come across a huge error that only affects entries made in the second week of December.

The first type of bug really must be eliminated before a program is going to be of any use to anyone.

And the second? Well, it is considered good programming sense to unearth these by testing a program thoroughly—and this includes trying the unexpected such as entering an 'impossible' sequence of keystrokes or input values. And this is where special error-trapping routines can be brought into use—a subject covered in the next part of this article.

ERROR MESSAGES

All home computers generate error messages of one kind or another, and you will quickly become used to these as you learn to use your computer.

These range from simple number and letter codes to full blown descriptions which leave you in little doubt about the exact nature of the fault—even if, on occasions, they do not exactly point to the fault itself!

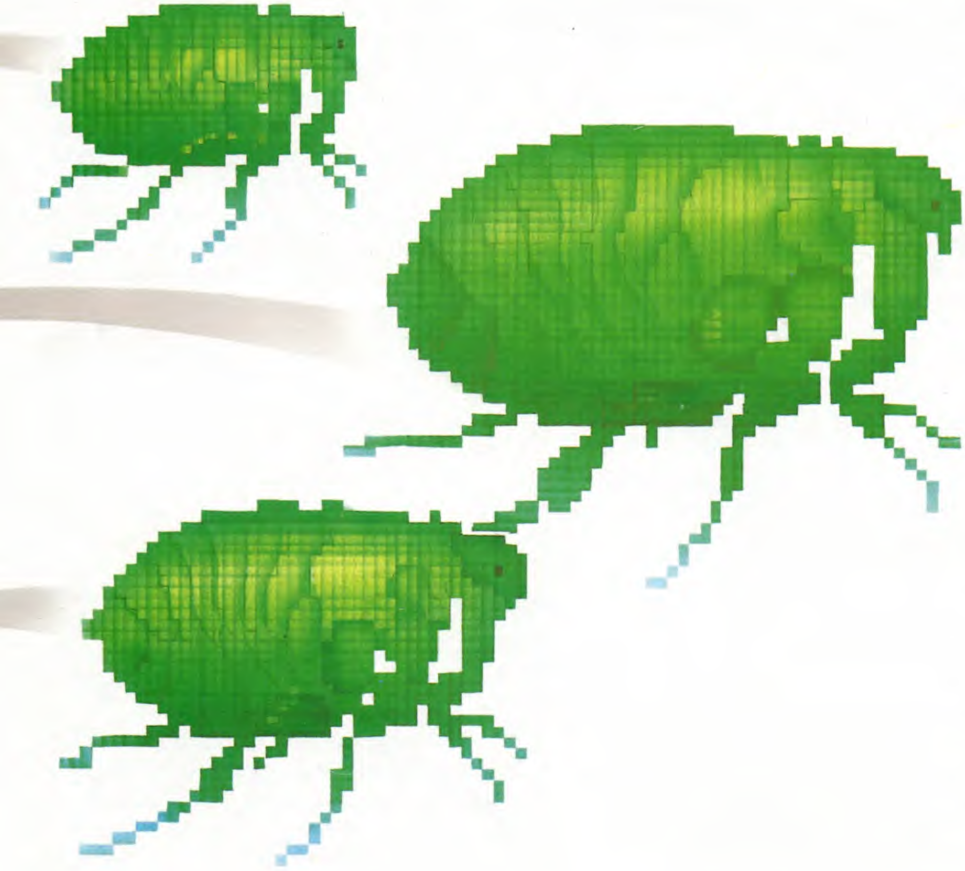
Full details of error messages can be found in your computer's manual, so use this to examine the meaning of those that you do not understand. Only then proceed with correcting the fault.

BUG TRACING

One important rule is to trace each error as it crops up. Do not leave a known bug within a program even if the program seems to RUN satisfactorily thereafterwards or if the bug only rarely affects it.

Otherwise there's just a chance that this 'nastie' will crop up at a critical time later on. The result could be a program crash which may mean a lot of extra keying-in time, or—worse—the loss of valuable data if the program

- THE IMPORTANCE OF TRACING EACH ERROR AS IT CROPS UP
- ERROR MESSAGES AND WHAT THEY SIGNIFY
- DIRECT AND INDIRECT LEADS
- AVOIDING COMMON ERRORS
- TAKING CARE OVER DATA STATEMENTS AND SPACES
- PROGRAM TO TEST YOUR BUG FINDING ABILITY



has been put to actual use. At the very least you will lose a valuable opportunity to correct the problem.

Tracing errors—'debugging'—can be a fearfully complicated task unless you try and isolate each problem in an orderly manner. In typing up a program list—or developing your own program—rather more than a single error is likely. But each error must be treated individually—first located then corrected—before moving onto the next.

Start by thinking simple. The simplest of errors is often amongst the most difficult to isolate and—contrary to what you might believe at the time—often the sole cause of difficulties in the proper RUNning of a program.

What you have to establish is a set routine for debugging your programs. There are some simple techniques and effective shortcuts.

To start with, quite a few error messages actually point directly to the line in which the error occurs. This includes the commonest of the lot—SYNTAX ERROR— and several others (see table). But many others are less obliging, indicating errors on lines that are in fact perfectly correct. For example, you might get a message like E Out of DATA, 1Ø:2 (this example is from the Spectrum, but the report on other machines is similar). This suggests that the error is in the second statement in Line 1Ø. In fact, the second statement in Line 1Ø would tell the computer to READ some DATA, which might appear in Line 2ØØ or even 1ØØØ or later. And it would be the DATA lines which contained the error, not Line 1Ø. This type of error is, understandably, much more difficult to trace for there's no clearcut indication of where the real problem lies.

With line-indicating error messages you know the error *must* be something to do with the way you have made an entry in a particular line within the program.

Start by LISTing the program from a point just before the line number suggested in the error message. It sometimes helps to LIST the line itself, separately, a little way from these lines if your computer has the facility to do this.

Check first of all that you have not made some sort of *literal error*—misspelt something or used a letter in place of a number (or vice versa). Be especially careful about spaces and punctuation. And look for missing characters—it's very easy to leave off a bracket in a sequence which makes use of several, for example. Look at keywords letter by letter rather than simply glance at them: it's all too easy to transpose characters within these.

This type of error (see below) is by far the most common cause of problems and is something that plagues beginner and expert alike.

PIECE BY PIECE

When you know which line contains an error, one method of isolating the problem statement in a line containing two is to place a STOP statement in a new line immediately after the line indicated by the error message. Then insert a REM at the very beginning of the last statement in the line and reRUN the program—obviously only if it didn't crash at this point before. If the syntax error no longer appears then you know the problem must lie somewhere in that last statement. You can use a similar method to track down an offending line if you know which of several it might be. Insert an extra line between each in turn, containing a STOP statement as detailed later.

But this method cannot safely be extended to multiple-statement lines as everything after the REM and before the next program line is ignored.

With lines containing multiple statements, break the line into its component parts and check each of these in isolation. Each statement can be split into a further line. This is a simple but useful ploy if things really do look foggy and confused.

ERROR LOCATION

The following is a list of error messages and reports for each machine. Where an entry is preceded by an asterisk (*), the error most likely does occur in the line whose number is indicated in the error message. Or else the error is clearly the immediate result of a direct entry or activity (such as SAVEing). There are always exceptions, particularly where a variable which is the cause of an error in one line has been assigned a value in a previously executed line. When keying in programs be especially careful to include space where indicated (in *INPUT* by using the symbol □)—but if in doubt, leave them out! On the Commodore remember to type out PRINT# in full to avoid a syntax error.

NEXT without FOR, Variable not found, Subscript wrong, * Out of memory, * Out of screen, Number too big, RETURN without GOSUB, * End of file, * STOP statement, * Invalid argument, Integer out of range, * Nonsense in BASIC, * BREAK—CONT repeats, Out of DATA, * Invalid file name, * No room for line, * STOP in INPUT, FOR without NEXT, * Invalid I/O device, * Invalid colour, * BREAK into program, * RAMtop no good, * Statement lost (cannot occur in program), * Invalid stream, FN without DEF, Parameter error, * Tape loading error.

BAD DATA, BAD SUBSCRIPT, * BREAK, * CAN'T CONTINUE, * DEVICE NOT PRESENT, DIVISION BY ZERO, * EXTRA IGNORED, * FILE NOT FOUND, FILE NOT OPEN, FILE OPEN, * FORMULA TOO COMPLEX, * ILLEGAL DIRECT, * ILLEGAL DEVICE NUMBER, ILLEGAL QUANTITY, * LOAD, * MISSING FILE NAME, NEXT WITHOUT FOR, NOT INPUT FILE, NOT OUTPUT FILE, OUT OF DATA, OUT OF MEMORY, OVERFLOW, REDIM'D ARRAY, * REDO FROM START, RETURN WITHOUT GOSUB, STRING TOO LONG, * ?SYNTAX, TOO MANY FILES, * TYPE MISMATCH, * UNDEF'D FUNCTION, * UNDEF'D STATEMENT, * VERIFY.

* Accuracy lost, * Arguments, * Array, * Bad call, * Bad DIM, * Bad hex, * Bad MODE, * Bad program, * Block?, * Byte, Can't match FOR, Channel, * Data?, * DIM space, Division by zero, $ range, * Eof, * Escape, Exp range, Failed at Ø, * File?, * FOR variable, * Header?, * Index, * LINE space, Log range, * Missing ,, * Missing ", * Missing), * Mistake, -ive root, No GOSUB, No FN, No FOR, No PROC, No REPEAT, No room, No such FN/PROC, * No such line, No such variable, * No TO, Not LOCAL, ON range, * ON syntax, Out of DATA, Out of range, * Silly, * Syntax, String too long, Subscript, * Syntax error, Too big, Too many FORs, Too many GOSUBs, Too many REPEATs, * Type mismatch.

/Ø, AO, BS, * CN, * DD, * DN, DS, FC, FD, FM, * ID, IE, * IO, LS, NF, NO, OD, OM, OS, OV, RG, * SN, * ST, * TM, * UL.

ReRUN the program to see which of the new lines in the group causes the error message and take it from there.

Very long individual statements present a different sort of problem, and perhaps the best way of tackling these is to determine the actual values of each part of it.

Again, presume you've made a silly error before reading too much into the situation. If possible, rewrite the offending program line to produce a number of stand-alone lines which can be treated individually. This is in itself one very good reason for leaving plenty of space between your line numbers.

Where you can't do this, mentally 'explode' the statement and ask yourself exactly what each entry is doing at that point in the program. Try to work out what values have been obtained for any variables in use at that point in the program.

With the syntax error the values of any variables in action at that point are irrelevant—they won't be the cause of the problem—and changes you wish to make to the line can be done even though this clears the variables.

If you are dealing with more obscure problems (see below) the actual variable value(s) can provide clues. You can use your computer to PRINT these out in direct (immediate) mode *before* any changes are made to the program.

While you're at it, check that the name of the variable is correct. For example, if you habitually use the variable D for a time delay loop, you may subconsciously introduce this when you come across a similar loop in a listing you are copying but which uses a different variable—a T, say.

OBSCURE LEADS

Many error messages do *not* give clear indication of the offending line number. Instead they merely indicate where an error has had some *effect* and so it can be difficult to pinpoint the cause.

But there are exceptions. As in the example above, for instance, DATA error messages are displayed in a line containing a statement which include READ when in fact the error itself is almost certainly in the DATA statement line it is trying to access.

Others are less obliging (see table). Most of these do, however, refer to errors which occur *before* the quoted line number in the *execution* of the program. This is an important distinction, because the error could in fact be in a subroutine well before or after the program line indicated in the error statement. In this way a variable may pick up an incorrect value long before it is spotted.

The best way to treat less obvious errors of this type is to examine the action of the program at the time. Start by PRINTing out the value of every variable—if you have a printer attached you could make a hard copy of these. But it's almost as easy to use the direct PRINT command or any allowable abbreviation followed by the variable name—such as PRINT V. Note down the values.

Now examine how the variables work in relation to each other. Again, you may find it helpful to split a long program line into smaller lines, each containing a single statement.

Trace back each variable from this point in the program to compare its real value (the figure you have noted) with what it should have been if the program had RUN correctly.

This can, unfortunately, be a long and difficult job in a lengthy program. Some of the donkey work can be removed by RUNning the program through various distinct stages—such as just before and just after a particular set of inputs—and note how the values of the variables change.

Unless you are timely with the use of the RUN/STOP or BREAK keys of your computer, it's advisable to introduce temporary lines containing the STOP statement within your program at these points. But note that this adjustment will clear memory and so the program has to be reRUN. This may not always be possible.

When the program STOPs, PRINT the variable values and then key in the instruction to continue to the next STOP.

COMMON ERRORS

The most common cause of program errors are mistakes made in entering code from printed listings.

Regardless of the cause, the particular problem here is confusing characters which look similar: typically lower case l with capital I or 1, and O with Ø. (In the latter respect, on the BBC, avoid programming in the Teletext mode where the difference seems marginal when displayed.)

Also easy to confuse—especially with poor quality listings—are the colon and semicolon, and the full point and comma. The semicolon is typically used in formatting the display, and accidental use of a colon—which means end of a statement—in a line such as this would simply result in a SYNTAX ERROR message:

```
95 INPUT "ENTER YOUR NAME": N$
```

But confusion over the full point and comma can be much more difficult to spot. Take these two lines, for instance:

```
1ØØØ DATA 12.4,6,7,8,13,1.7
1ØØØ DATA 12.4,6,7,8.13,1.7
```

Both *could* be correct—and certainly would appear so even on fairly close scrutiny. There are six items of DATA in the first, but only five in the second.

This may provide some sort of clue if all the other DATA statements are restricted to a set number of items each. This, incidentally, is a simple method of providing at least one check.

If an error message displayed OUT OF DATA then there are not enough DATA items and so the second line would be incorrect. The presence of too many DATA items does not cause an error message, but simply assigns the wrong values to the READ variable. Or it may assign the wrong value to *some* of the variables if another DATA line is short . . .

An error message may also result either at the READ line or a calculation line if the wrong *type* of value is assigned to the READ variable. And the use of a O rather than Ø—or vice versa—can have just this effect.

SPACES

Spaces have to be left in some places in a program but most definitely omitted in others—the problem is often recognising that something as 'transparent' as this is the cause of a problem.

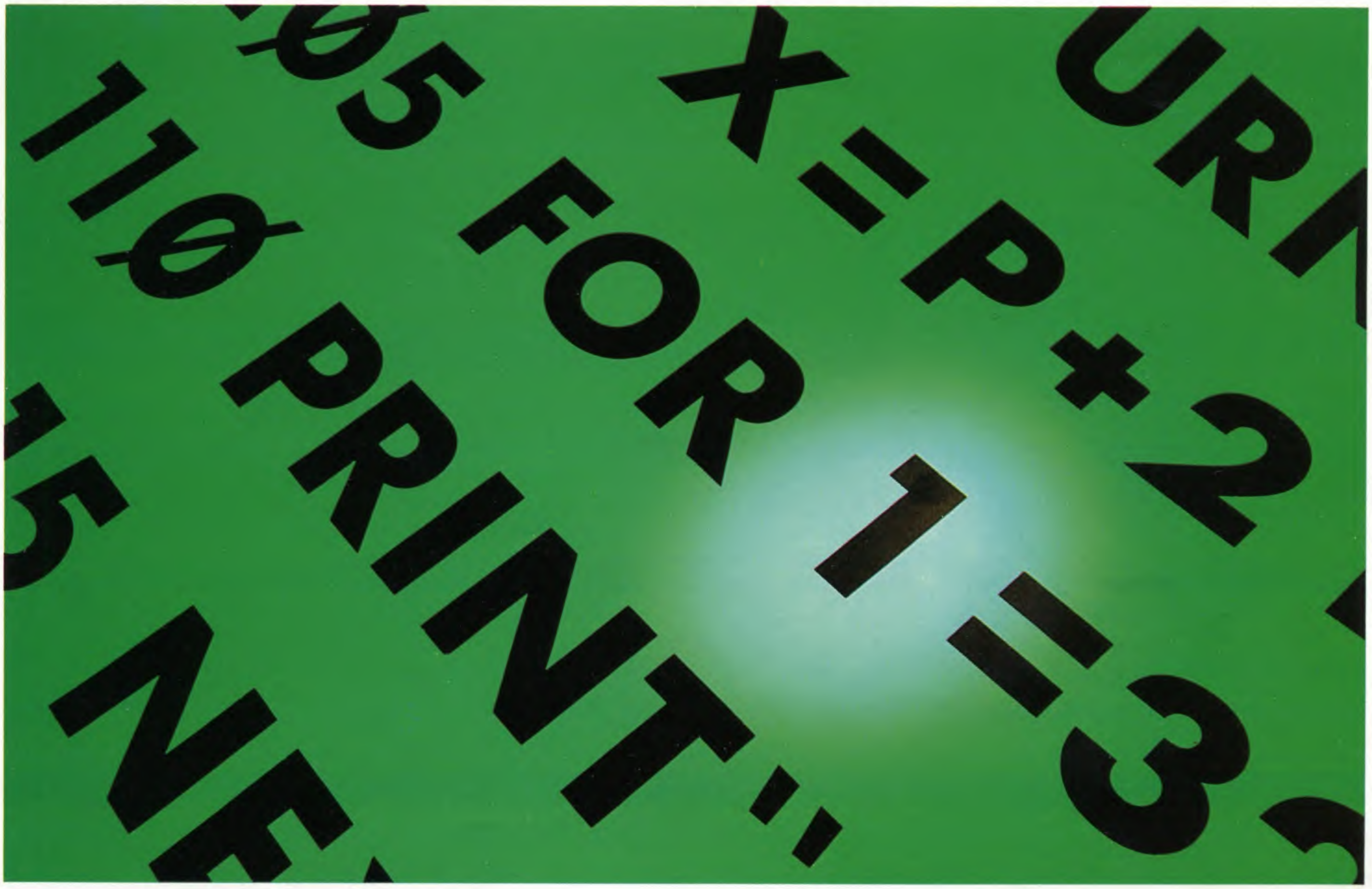
Spaces which are used for cosmetic reasons, for example to separate one PRINTed statement from another or to make listings a little clearer, do not cause errors if omitted. But suspect that this is what has happened if your screen displays look squashed up or parts are overwritten by others.

Both cosmetic and necessary spaces are indicated by the symbol □ in *INPUT* listings to avoid confusing the number of spaces you need to type in. Even a computer print out can be confusing . . . is it 11 or 12 spaces?

An error message results if you accidentally *include* a space between certain keywords and the bracketed argument which follows. TAB(n) is correct but TAB□(n) is not. The requirements in this respect differ from one computer to another but it's worthwhile checking these points if all else fails.

OTHER FAULTS

Each computer has its own peculiar types of bug. Often these relate to imperfections—rather than true faults—in the operating system or the hardware itself. Quite often the failing is in BASIC itself and a number of keywords may not behave properly under certain circumstances. Some of the more common ones are detailed here, and they will be pointed out from time to time in *INPUT* where they might cause unexpected problems if care is not taken.



BUG SWATTING

Here's a test of your bug-finding ability and a chance to put what you've learned into practice. The programs below are full of bugs—the sort you might introduce when you copy in a listing from a magazine or when you try to adapt a program without fully understanding what it does. They range from simple typing errors and syntax errors to errors in the structure of the program itself.

The correct version of the program appeared on page 155 of *INPUT*, apart from an extra line that's been added here to make the program re-run. But don't look yet until you've had a proper go at finding the bugs yourself.

The program is a very short example of how you might assign objects to rooms in an adventure game. The list of objects is READ from a DATA list into one array while the numbers of the rooms is stored in another array. The main part of the program—Lines 7Ø to 1ØØ—assigns one object randomly to each room and makes sure that all the objects are used and there is only one per room. Line 11Ø simply PRINTs out the result. At least that is what the program *should* do if it was written correctly!

You'll probably be able to spot a lot of the errors straight away just by reading through the program. Try to put right as many bugs as you can in this way and then, when it's as 'clean' as you can make it, type it in to your computer and see if it will RUN. No doubt there'll still be a few more errors lurking there that you hadn't spotted at first.

If you really get stuck then look back to page 155 although you'll have to work out the correct version of the last line yourself. But be careful not to introduce any new bugs yourself when you're correcting the others!

```
1Ø LET g=14
2Ø DIM a$ (g,7): DIM a(g)
3Ø FOR z=1 TO g
4Ø READ a$(g)
5Ø LET a (z)=z
7Ø FOR x=g TO 1 STEP –I
8Ø LET q=INT (RND*x)+1
9Ø LET t=a(x): LET a(x)=a(q):
   LET a(q)=T
1ØØ NEXT x
11Ø FOR t=1 TO g: PRINT “Room ;
   t; has a”;a$(a(t)): NEXT t
12Ø DATA rope,“sword”,“spanner”,
   “knife”,“gun”,“key”“torch”,
   “car”,“whip”,“wand”,“bomb”,
   “book”,“model ship”,“robot”
13Ø GOTO 1Ø
```

```
1Ø LET G=14 : PRINT CHR$ 147
2Ø DIM A$(G),A (G)
3Ø FOR Z=1TØG
4Ø READ A$(G)
5Ø A(Z)=Z
7Ø FOR X=GTO1 STEP –I
8Ø Q=INT(RND (1)*X)+1
9Ø T=A(X): A(X)=A(Q); A(Q)=t
1ØØ NEXT X
11Ø FOR T=1TOG:PRINT “ROOM”;T
   “HAS A” A$(A)(T)):NEXT T
12Ø DATE ROPE,SWORD,SPANNER,
   KNIFE,GUN,KEY,TORCH,CAR,
   WHIP,WAND,BOMB,BOOK,MODEL
   SHIP,ROBOT
13Ø GOTO 1Ø
```

```
1Ø LET G=14
2Ø DIM A$(G),A (G)
3Ø FOR Z=1TØG
4Ø READ A$(G)
5Ø A(Z)=Z
7Ø FOR X=GTO1 STEP –I
8Ø Q=RND (X)
9Ø T=A(X): A(X)=A(Q); A(Q)=t
1ØØ NEXT X
11Ø FOR T=1TOG:PRINT “ROOM”;T
   “HAS A” A$(A)(T)):NEXT T
12Ø DATE ROPE,SWORD,SPANNER,
   KNIFE,GUN,KEY,TORCH,CAR,
   WHIP,WAND,BOMB,BOOK,MODEL
   SHIP,ROBOT
13Ø GOTO 1Ø
```

HOW TO MERGE PROGRAMS

- WHY MERGE PROGRAMS?
- ADDING EXTRA SUBROUTINES
- STRINGING SEVERAL PROGRAMS TOGETHER
- HOW TO MERGE

Why spend precious computing time rekeying programs that you have already typed and tested, when you can enter a few commands and let your micro do the work?

Keying in even a short program can be tedious—especially if you are less than an expert typist. Not only is it laborious, but there is always a risk of introducing errors—either in copying wrongly or by mis-keying. So any method by which you can save yourself unnecessary typing is well worth using.

There is, unfortunately, no magic answer—anything that you don't LOAD from a prerecorded tape has to be typed into the computer at some time or another. But you can often economize on the amount of new material you have to type by editing and reusing an earlier program—or by bringing together two or more programs to form one.

There are basically two different ways of combining programs. *Appending* is where the line numbers of one program are all bigger than those in the other, and they can simply be joined end to end. *Merging* is a more complicated process in which two programs can be knitted together in spite of having similar line numbers. For the purposes of this article, both methods will be treated together.

WHEN TO MERGE

Merging is essentially an aid to program development. Imagine that you have written a program that does not work as you expected, or that you wish to change an existing program to achieve a different task. Your best course would be to SAVE the program on tape or disk, then you can continue to work on it further without fear of ruining the original. After further development, the program might be twice as long as the version you originally SAVEd, or it might have gained several different sections. In any event, both versions may have elements that you wish to keep, but with so many differences that they would be tedious to key separately. And you have the further problem that—short of working it all out the long way on paper—you need to have both versions in the memory at once so you can combine them. This is a case for merging.

A similar need to merge will arise when you wish to incorporate existing subroutines, procedures or functions into a new program. These could be long sections of code that play a tune, animate a piece of graphics or even draw a graph of data that has been calculated by your program.

Most programmers have a whole store of such routines that they have built up over the years, and it is always a good idea to do the same yourself. Then you can merely drop them into any program as required. Once these sections have been tested and SAVEd, you would not wish to retype them every time, but

you can incorporate them by merging.

Another important use for merging is when you wish to string short programs one after the other, so that they run consecutively. There are all sorts of uses for this, but perhaps the facility is most useful if you use your micro to display information, such as messages, or interesting graphics merely for pleasure. For example, you could have the first program print a greeting, then the second could build up a colourful picture, and so on. The last program could end with a line which sends the micro back to the first line of the first program, giving a continuous display.

HOW TO MERGE

Each micro has its own method of merging programs. Some have a MERGE command, which ensures that any program already in memory is retained while another is loaded from tape or disk. Others require you to enter a few lines of code, or to allocate space in the memory for the second program. Whatever the method used, you must arrange the line numbers so that the merge is as required.

The ZX81 is a special case. It does not have a merge command, and you cannot easily load more than one program in memory. See the Q & A box for more information.

Merging is made simple on the Spectrum by the MERGE command. But you must be careful with the line numbering of the two programs. For example, to merge a 20-line subroutine with a larger program, you could leave a gap in the numbering. The program would be numbered from, say, 1Ø to 5Ø then 3ØØ to 1ØØØ, and the subroutine from 6Ø to 25Ø.

Alternatively, you could give the subroutine line numbers that are all larger than those of the program, so that it appears at the end of the program. With a little care, however, you can arrange for some lines of one program to overwrite those of another, and add new lines.

Enter and RUN this program to see squares printed at random positions in two rectangles. In a moment you will see how to merge it with another program:

```
1Ø BORDER Ø: INK 9
5Ø PAPER Ø
6Ø CLS
7Ø FOR n=1 TO 4ØØ
8Ø LET x=INT (RND*32)
9Ø LET y=INT (RND*22)
1ØØ PAPER 7: IF x>8 AND x<24 AND y>6
   AND y<16 THEN PAPER 4
11Ø PRINT AT y,x;"□"
12Ø NEXT n
```

SAVE this program (as SQ1, say). Do not disconnect the tape or disk—you will be LOADing the program back into the computer almost at once. Of course, you could equally well use a program that you SAVEd some time before. All you have to do is to connect the recorder to the computer first. Enter NEW to clear the memory before typing in the next program:

```
1Ø BORDER Ø: PAPER Ø: INK 9: CLS
3Ø FOR t=1 TO 5
4Ø INK INT (RND*6)+2
6Ø PRINT AT t,Ø; "THIS IS A TEST"
13Ø NEXT t
```

This program is a simple example which PRINTs a message (Line 6Ø) five times, but it could be a much longer program into which the other is to be merged. Enter MERGE“” to merge the two programs together. The stored program will not be merged into the one above. LIST the new program and notice that Lines 1Ø and 6Ø have been overwritten, while Lines 5Ø, and 7Ø to 12Ø have been added. RUN the program to see the squares program (SQ1) executed five times.

There is no facility on the Spectrum to renumber a program, so if you wish one program to appear at the end of another, you must first edit it to adjust the line numbers.

You could write a program to merge programs on Commodore micros, but it would not be a simple exercise. It is far better to make use of two simple commands that allow you to LOAD one program on top of another, without erasing either of them. The program below is not suitable for the Vic 20 but the method of merging is the same. So try it out on some of your own programs instead. To see how this works, enter and RUN the first program.

```
10 FOR T=1 TO 5
20 CC=RND(1)*6+1
30 POKE 53281,0
40 PRINT "♥"
50 FOR N=1 TO 300
60 X=INT(RND(1)*40)
70 Y=INT(RND(1)*25)
80 IF (X<8 OR X>32) OR (Y<5 OR Y>19)
   THEN C=CC:GOTO 100
90 C=7
100 POKE 1024+Y*40+X,160:
   POKE 55296+Y*40+X,C
110 NEXT N,T
```

This program prints squares at random positions across a rectangular area on the screen.

SAVE the program (as SQ1, say) but do not disconnect the recording unit, since you will be LOADing the program back into the computer in a moment. On the Vic 20 start off by PEEKing two memory locations; and remember the numbers (call them A and B) because you'll need them later on:

```
PRINT PEEK(43)
PRINT PEEK(44)
```

Then enter the next line.

```
POKE 43,PEEK(45)-2:POKE 44,PEEK(46)
```

This command finds out where in the memory the program ends and sets that position as the loading address of the next program so the programs follow on one after the other.

Now type NEW and enter the second program or LOAD it from tape:

```
200 FOR P=1 TO 30
210 PRINT "♥"
220 FOR T=1 TO 34
230 PRINT "▤▣▣▩"TAB(T)
   "□♣□□π"
240 FOR Z=1 TO 30:NEXT Z
250 NEXT T,P
```

This program shows a man and a dog moving across the screen. Notice that the line numbers are all larger than those of the SAVEd program. This is essential, because there is no standard facility to renumber program lines on Commodores.

Now enter the next command, which tells the micro where in memory the second program begins. On the Commodore 64 use:

```
POKE 43,1:POKE 44,8
```

and on the Vic 20 use:

```
POKE 43,A:POKE 44,B
```

where A and B are the numbers you found earlier. The two programs will now run one after the other, they reside in memory together and can be treated as one program.

Finally, enter LIST and the merged listing below will appear.

```
10 FOR T=1 TO 5
20 CC=RND(1)*6+1
30 POKE 53281,0
40 PRINT "♥"
50 FOR N=1 TO 300
60 X=INT(RND(1)*40)
70 Y=INT(RND(1)*25)
80 IF (X<8 OR X>32) OR
   (Y<5 OR Y>19)
   THEN C=CC:GOTO 100
90 C=7
100 POKE 1024+Y*40+X,160:
   POKE 55296+Y*40+X,C
110 NEXT N,T
200 FOR P=1 TO 30
210 PR!NT "♥"
220 FOR T=1 TO 34
230 PRINT "▤▣▣▩"TAB(T)
   "□♣□□π"
240 FOR Z=1 TO 30:NEXT Z
250 NEXT T,P
```

Microtip

Renumbering

If your micro has a RENUMBER command, you can use it to change the line numbers so they start at any number and increase in any regular step. The Spectrum and Commodore micros do not have this facility, but you can buy utility programs that provide it, together with many other useful features, which are explained in the literature that comes with the package. If you do not have one of these programs, then you must renumber line by line manually, making note of how numbers mentioned at GOTO, GOSUB and ON will have to be changed. Work out the changes and jot them down on paper, to avoid confusion.

There is no MERGE command on Acorn micros, but programs can be merged simply by either of two methods.

METHOD ONE

To see how the first method works, key in and RUN the program below. In a moment, you'll see how to merge this with another:

```
1Ø MODE 1
2Ø VDU23;82Ø2;Ø;Ø;Ø;
4Ø C=RND(3)+127
5Ø COLOUR128
6Ø CLS
7Ø FOR N=1 TO 8ØØ
8Ø X=RND(4Ø)-1
9Ø Y=RND(3Ø)-1
1ØØ IF (X<8 OR X>32) OR (Y<7 OR
   Y>21) THEN COLOUR C ELSE COLOUR
   131
11Ø PRINT TAB(X,Y);"□"
12Ø NEXT
```

This program prints white squares at random positions within a rectangular area on the screen, and coloured squares at random positions around the white area. Before you can merge this program with another, you need to save it as a file of ASCII codes (see page 314). Have your cassette recorder ready, then key in the following lines, one at a time, and respond to the prompts on the screen.

```
*SPOOL "SQ1"
LIST
*SPOOL
```

The first line tells the micro to open a file (called SQ1) on cassette or disk. The second line lists the program on the screen and sends the same data to the file. In this way an ASCII file of SQ1 is created.

Do not disconnect the tape or disk unit. You will shortly be asked to LOAD the program back into the computer and merge it with a new one. But obviously, the method would equally well apply to anything you have on file and ready to LOAD. You just have to set up the tape or disk unit first.

The next step is to prepare your main program. This can have any line numbers, but if the same number appears in both programs, one will overwrite the other. Enter NEW, then type in the next program.

```
1Ø MODE 1
3Ø FOR T=1 TO 5
4Ø COLOUR RND (2)+127
6Ø PRINT "THIS IS A TEST"
13Ø NEXT
```

This prints a message (Line 6Ø) on five lines of random coloured background. Although this is only a short, simple program, the principle holds true for any program, provided that the combined length is not more than the micro's memory can hold. And although you entered this program directly, the method would also work for any program that you had stored on a tape or disk and then LOADed in the usual way.

Now you are ready to merge the saved program into the one in memory. Make sure your tape or disk unit is connected, then enter the next line and respond to the prompts:

```
*EXEC "SQ1"
```

When you LIST this program, it will be as below.

```
1Ø MODE 1
2Ø VDU23;82Ø2;Ø;Ø;Ø;
3Ø FOR T=1 TO 5
4Ø C=RND(3)+127
5Ø COLOUR128
6Ø CLS
7Ø FOR N=1 TO 8ØØ
8Ø X=RND(4Ø)-1
9Ø Y=RND(3Ø)-1
1ØØ IF (X<8 OR X>32) OR (Y<7 OR
   Y>21) THEN COLOUR C ELSE COLOUR
   131
11Ø PRINT TAB(X,Y);"□"
12Ø NEXT
13Ø NEXT
```

The stored program has been added to the one in memory, with two differences: the '2' in Line 4Ø has been changed to '3', and Line 6Ø has been changed to CLS. RUN the program, and it will print the picture of squares five times, changing colours each time.

Notice that all three programs work satisfactorily, but that this is not always the case. In fact, you need to be particularly careful with the line numbers to ensure the commands are in the correct sequence. Usually, you will need to merge unfinished programs, then arrange the line numbers.

METHOD TWO

The second method of merging requires you to renumber one program so that the line numbers are all larger than those of the program in memory. To do this, make use of the RENUMBER command. Enter RENUMBER 15Ø then LIST the program you have in memory. Notice how the line numbers have been changed. Where they once read from 1Ø through to 13Ø, they now read from 15Ø to 27Ø. SAVE this program (as SQ2, say), then enter NEW and the next program:

```
1Ø MODE 1
2Ø VDU19,Ø,2,Ø,Ø,Ø
3Ø VDU23;82Ø2;Ø;Ø;Ø;
4Ø VDU23,224,28,28,8,126,8,24,36,72
5Ø VDU23,225,Ø,Ø,Ø,131,252,6Ø,36,36
6Ø FOR P=1 TO 3Ø
7Ø CLS
8Ø FOR T=Ø TO 34
9Ø VDU31, T,5,32,224,32,32,225
1ØØ A=INKEY(2)
11Ø NEXT
12Ø NEXT
```

RUN the program to see a man and a dog moving across the screen. The program you stored earlier, SQ2, can now be appended to this program by telling the computer where in memory it should be loaded.

To find out where in memory the program ends, enter the next line:

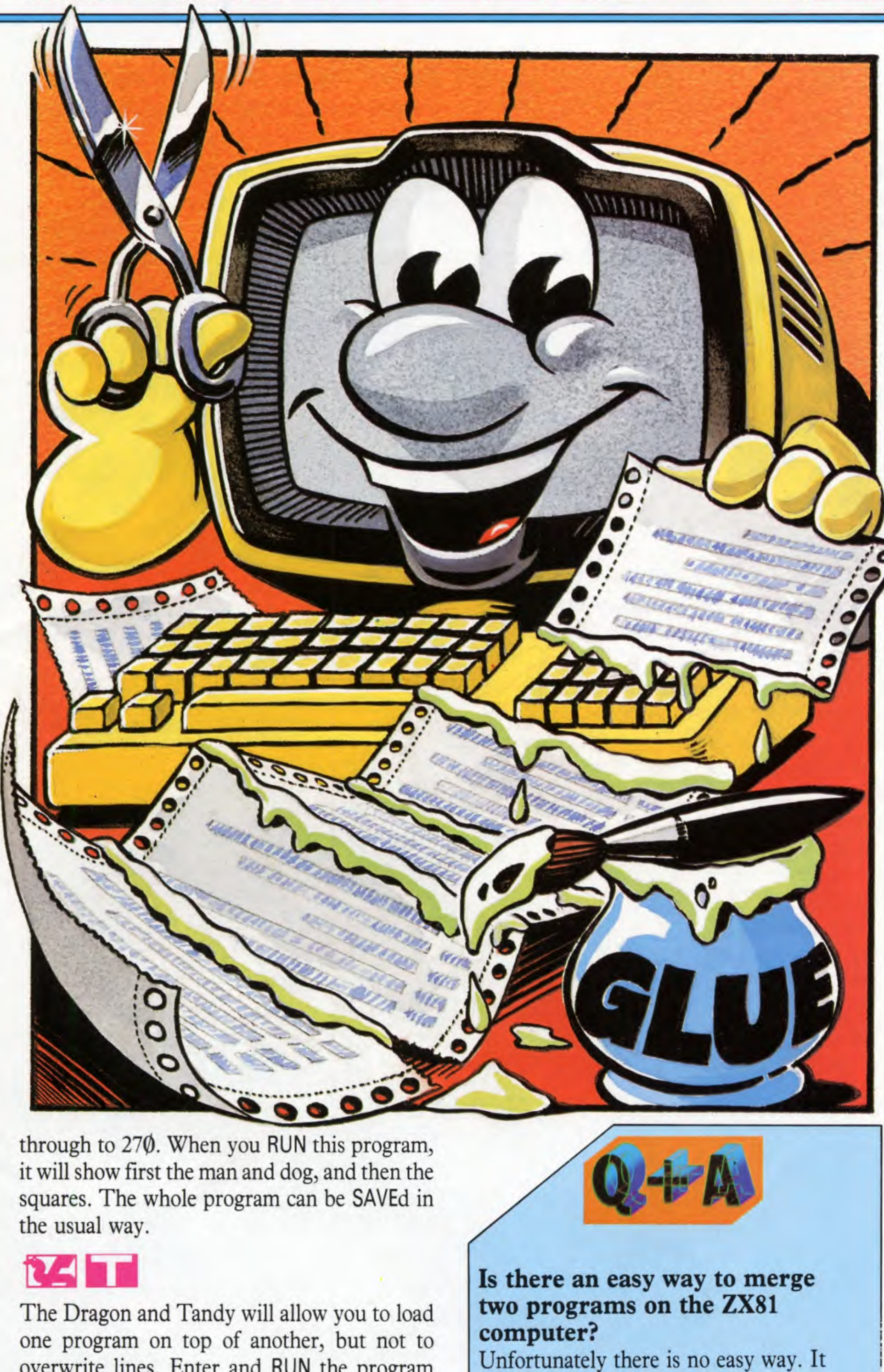
```
PRINT ~TOP-2
```

This causes a three or four-digit memory address to be printed on the screen. Now LOAD the SAVEd program (SQ2) to this address by entering the next line.

```
*LOAD "SQ2" XXXX
```

You must begin the command with an asterisk, and finish with the memory address that was printed on the screen (XXXX).

When the program has been loaded, enter END followed by RETURN. This last command is crucial; it enables the computer to reposition its pointers, which guide it through the program. The commonest reason for this method of merging failing to work, is because of forgetting to enter END after loading the second program. Now LIST the program and notice that the line numbers run from 1Ø



through to 27Ø. When you RUN this program, it will show first the man and dog, and then the squares. The whole program can be SAVEd in the usual way.

The Dragon and Tandy will allow you to load one program on top of another, but not to overwrite lines. Enter and RUN the program below, which prints coloured squares at random positions within rectangular areas of the screen.

```
10 PCLEAR4
2Ø FOR T=1 TO 5
3Ø CLSØ
4Ø FORN=1 TO 4ØØ
5Ø X=RND(32)−1
6Ø Y=RND(16)−1
7Ø IF X<6 OR X>25 OR Y<4 OR Y>11
   THEN C=175 ELSE C=255
8Ø POKE 1Ø24+Y*32+X,C
9Ø NEXT
1ØØ NEXT
```

Now SAVE the program (as SQ1, say), so that it can be LOADed back into memory at any time. This program could be a long subroutine or part of an unfinished program. Ordinarily, you would connect up the recording unit ready to LOAD it back into the computer, but now just leave the recorder connected.

Type NEW and enter the next program:

```
1Ø PCLEAR4
2Ø CLS
3Ø PRINT@192,STRING$(32,CHR$
   (236));
4Ø PRINT“□WELCOME TO A COLOURFUL
   DISPLAY”
5Ø PRINTSTRING$(32,CHR$(163))
```

It prints a message (Line 4Ø). To LOAD the program you have SAVEd, enter the following lines.

```
POKE 25, PEEK(27)
POKE 26, PEEK(28)−2
```

Next LOAD the program you have SAVEd (SQ1) then enter these two POKEs:

```
POKE 25, 3Ø
POKE 26, 1
```

When you LIST the new program, it will appear with line numbers from 1Ø to 5Ø and from 1Ø to 1ØØ. Enter RENUM to renumber the program, then LIST it again to give something like this:

```
1Ø PCLEAR4
2Ø CLS
3Ø PRINT@192,STRING$(32,CHR$
   (236));
4Ø PRINT“WELCOME TO A COLOURFUL
   DISPLAY”;
5Ø PRINTSTRING$(32,CHR$(163))
6Ø PCLEAR4
7Ø FOR T=1 TO 5
8Ø CLSØ
9Ø FORN=1 TO 4ØØ
1ØØ X=RND(32)−1
1ØØ Y=RND(16)−1
12Ø IF X<6 OR X>25 OR Y<4 OR Y>11
   THEN C=175 ELSE C=255
13Ø POKE 1Ø24+Y*32+X,C
14Ø NEXT
15Ø NEXT
```

The first part of the program is executed rapidly, followed by the second part. To separate them, change Line 6Ø as below.

```
6Ø FOR K=1 TO 2ØØØ:NEXT
```

Q+A

Is there an easy way to merge two programs on the ZX81 computer?

Unfortunately there is no easy way. It can be done in machine code but the program is quite complicated.

The ZX81 doesn't have a merge command and there is no way in BASIC to LOAD two programs into memory at the same time. This is because all BASIC programs are LOADed into the computer at the same address. So if you try to LOAD two programs, the second always overwrites the first.

A MOVING ADVENTURE

Part of the fun of playing adventure games is exploring a strange world without straying from home. We look at how the adventurer can set off on his explorations

Now that you have typed in a program containing all the location descriptions, you'll want the adventurer to be able to explore your world—moving from one location to the next. You need to be able to assess the possible moves at any point, and to make a choice based on your judgement and the clues which you have picked up as the adventure progresses.

This time, to expand your developing adventure program, you'll see routines that will pick out the correct location description, and display it, along with the possible exits. The player will then be asked to type in a response, and you'll learn how to write a section of program which moves the player through the world of the adventure according to the choice selected.

SETTING OFF

The first thing the computer needs to know is where the adventurer is at any moment during play. To do this, the program allows it to keep track of the whereabouts of the adventurer with a variable L—for location. The variable's value is changed to equal that of the current location at each move.

To start the adventure, then, you have to tell the computer which location you want the adventurer to be at initially.

The first section of a program to do this looks like this. LOAD the section of program you typed in last time, and add the new lines:

```
100 CLS: LET DA=Ø: LET TA=Ø:
   LET LA=Ø
27Ø REM **START POSITION**
28Ø LET L=15
29Ø GOTO 33Ø
```

```
27Ø REM ** START POSITION **
28Ø L=15
29Ø GOTO 33Ø
```

```
1Ø TA=Ø:LA=Ø
1ØØ MODE 6
27Ø REM**START POSITION**
28Ø L=15
29Ø GOTO 33Ø
```

15 is the location for the gate to the hidden city. If you want to start the adventure at a different location just change the value of L. In a moment you'll see how to adjust the value of L during play to suit a new location—but before the adventurer can move, the player will need to instruct the computer where to go next.

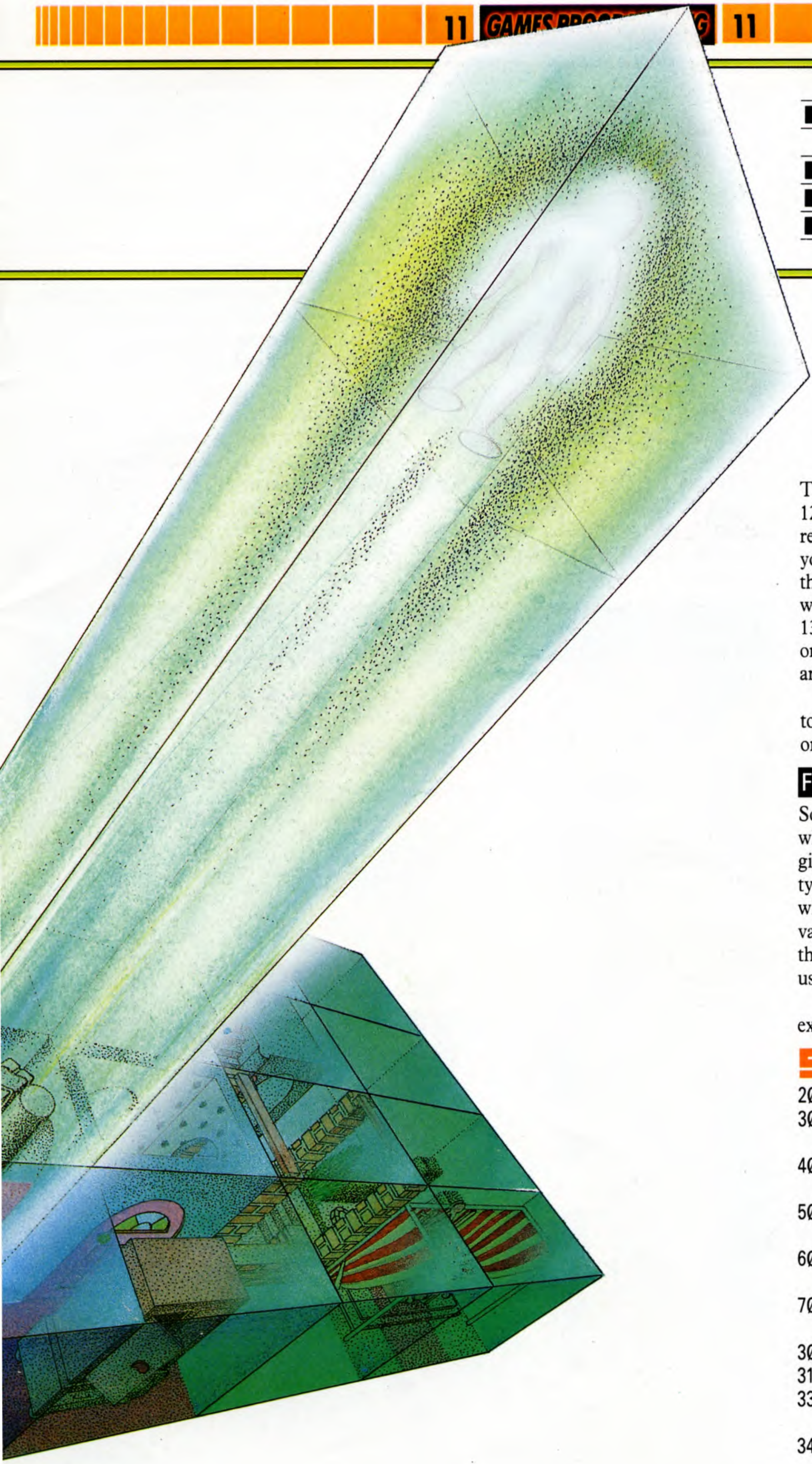
RESPONSES

So that the computer will be able to understand and to act correctly upon your responses, you must give the machine a list of the words it can recognise.

At this stage of development it only need recognise the four directions so that the adventure world can be explored. This can be done with an array R$, which holds the DATA for each direction response.

```
11Ø REM **SET UP ARRAYS FOR
   RESPONSES**
12Ø DIM R$ (19,4Ø): DIM R(19)
13Ø FOR K=1 TO 4: READ
   R$(K), R(K): NEXT K
15Ø DATA "NORTH", 1,
   "SOUTH", 1, "EAST", 1,
   "WEST", 1
```

```
11Ø REM**SET UP ARRAYS FOR
   RESPONSES**
12Ø DIM R$(19),R(19)
13Ø FOR K=1 TO 4:READ R$(K), R(K):NEXT
15Ø DATA NORTH, 1, SOUTH, 1, EAST, 1,
   WEST, 1
```



- DISPLAYING THE LOCATION DESCRIPTIONS AND DIRECTIONS
- HANDLING INSTRUCTIONS
- MOVING ROUND THE ADVENTURE
- BLOCKING IMPOSSIBLE MOVES

The arrays have been DIMensioned in Line 12Ø so that they will be able to hold all the responses needed for the game. At this stage you are only needing to use the directions, so the first four elements of array R$ and array R will be used. The FOR ... NEXT loop in Line 13Ø, READing into both R$ and R, goes from one to four. The directions, and their numbers are in Line 15Ø as DATA.

But obviously, this information is of no use to the player unless the computer also tells him or her where they are first.

FINDING A LOCATION

So that the adventurers can keep track of where they are after each move, they need to be given a location description. You've already typed these in, so you'll need a routine which will pick out the description which matches the value of L—the location number. This is where the REM lines you typed in last time will be useful.

Spectrum owners will have to type in one extra routine at this stage:

S

```
20 DIM G(11,4): POKE 23658,8
30 FOR N=1 TO 4: FOR M=1 TO 11: READ
   G(M,N): NEXT M: NEXT N
40 DATA Ø,Ø,Ø,5Ø2Ø,Ø,Ø,5Ø5Ø,
   5Ø8Ø,Ø,511Ø,Ø
50 DATA 514Ø,Ø,Ø,518Ø,521Ø,
   524Ø,527Ø,53ØØ,Ø,Ø,Ø
6Ø DATA Ø,533Ø,Ø,536Ø,Ø,Ø,Ø,
   Ø,Ø,Ø,Ø
7Ø DATA 1Ø1Ø,115Ø,124Ø,131Ø,141Ø,
   146Ø,15ØØ,136Ø,1Ø8Ø,155Ø,311Ø
3ØØ REM **FIND LOCATION**
31Ø CLS
33Ø IF L<11 THEN GOSUB G(L,1): GOTO
   4ØØ
34Ø IF L<21 THEN GOSUB G(L−1Ø,2):
   GOTO 4ØØ
35Ø IF L<26 THEN GOSUB G(L−2Ø,3)
```

```
300 REM ** FIND LOCATION**
310 CLS
330 IF L<11 THEN ON L
    GOSUB Ø,Ø,Ø,5Ø2Ø,Ø,Ø,5Ø5Ø,
    5Ø8Ø,Ø,511Ø:GOTO 4ØØ
340 IF L<21 THEN ON L−1Ø
    GOSUB 514Ø,Ø,Ø,518Ø,521Ø,
    524Ø,527Ø,53ØØ,Ø,Ø:GOTO 4ØØ
350 IF L<26 THEN ON L−2Ø
    GOSUB Ø,533Ø,Ø,536Ø
```

On the Commodore and Vic change Line 31Ø:

```
310 PRINT"□"
```

Before you can write this kind of routine you must be sure of the number of each location description. Starting with location 1, make a list of the line numbers of each description. If there is no description for a particular location, then write Ø. In this adventure, there are no descriptions for locations 1, 2 and 3, but there is one for location 4.

Now that you have the list of line numbers, you can begin writing the routine. On all the machines except the Spectrum, this is a simple sequence of operations which checks the value of L, then uses ON ... GOSUB. In the case of the Spectrum, which does not have this command, the line numbers should be fed into an array as in the example above.

In the Dragon, Tandy, Acorn and Commodore programs, the list of line numbers is in Lines 33Ø to 35Ø, starting at location 1 at the beginning of Line 33Ø, and ending with location 24 at the end of Line 35Ø.

The Spectrum, on the other hand, uses the value of L to pick out an element from the array filled by Lines 2Ø and 3Ø. You may notice that there are a few more numbers in the array than there are locations in the adventure. All the extra figures are noughts, so they have no effect on the operation of the program, although they are needed because the array will be used for calling other parts of the program—you'll see the additions to the array in a later part of Games Programming.

One last point for Spectrum users: the POKE in Line 2Ø merely sets the machine in upper case so that there are no problems matching the adventurer's input to the responses in R$.

DISPLAYING DIRECTIONS

In addition to the location descriptions, the adventurer will want to know which exits there are. It is not possible to move in all directions from each location, so the program will need to check this from the information contained in the location description—the variables N,E,S,W. The next section of program will tell the adventurer which directions are possible:

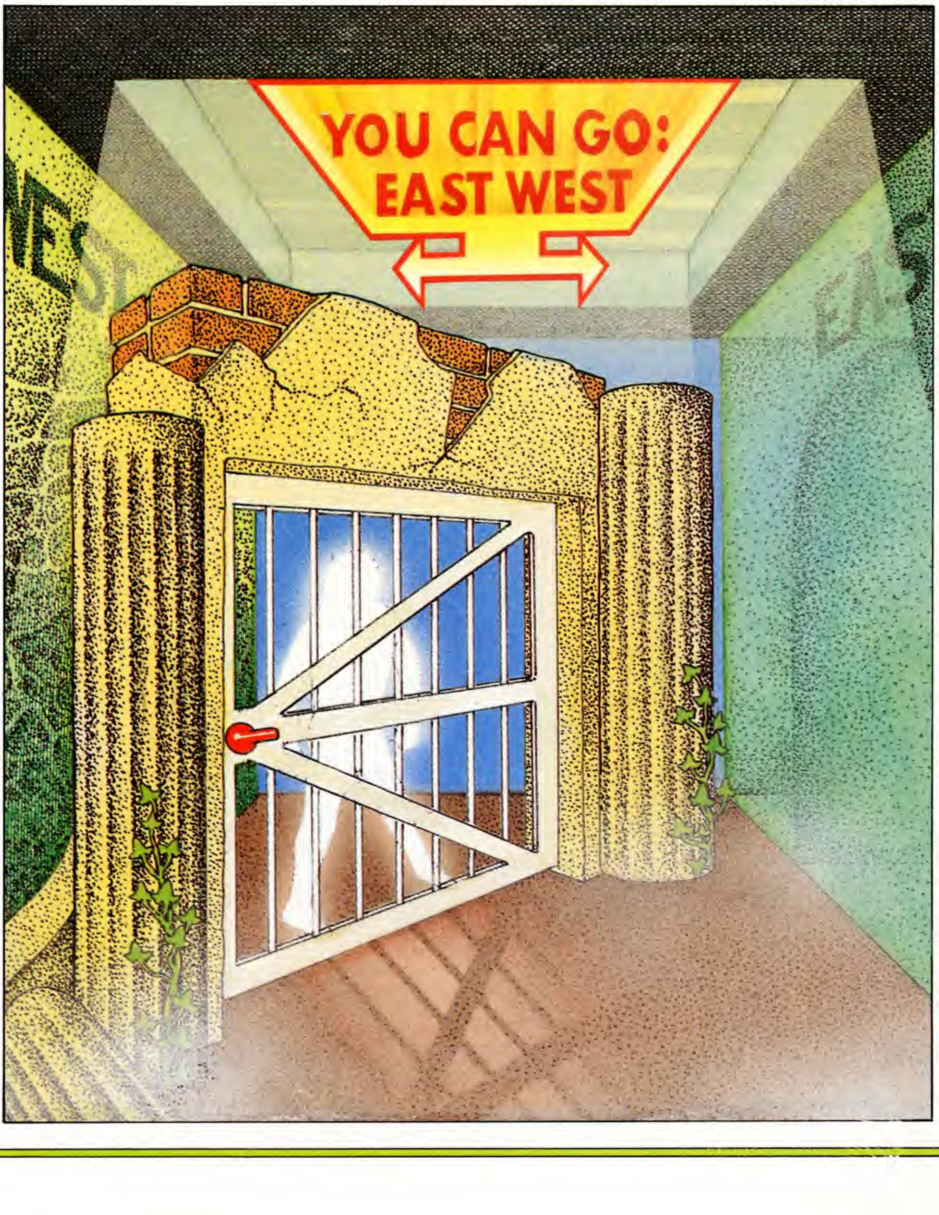
```
390 REM **DISPLAY DIRECTIONS**
400 IF DA< >1 THEN PRINT '"YOU CAN
    GO ";
410 IF N>Ø THEN PRINT TAB 11;
    "NORTH"
420 IF E>Ø THEN PRINT TAB 11;
    "EAST"
430 IF S>Ø THEN PRINT TAB 11;
    "SOUTH"
440 IF W>Ø THEN PRINT TAB 11;
    "WEST"
```

```
390 REM ** DISPLAY DIRECTION **
400 IF L< >11 OR (LA=1 AND
    OB(6)=−1) THEN PRINT:PRINT "◣YOU
    CAN GO π";:GOTO 41Ø
405 GOTO 46Ø
410 IF N>Ø THEN PRINT TAB(11);
    "NORTH"
420 IF E>Ø THEN PRINT TAB(11);"EAST"
430 IF S>Ø THEN PRINT TAB(11);"SOUTH"
440 IF W>Ø THEN PRINT TAB(11);"WEST"
```

```
390 REM **DISPLAY DIRECTIONS**
400 IF L< >11 OR (LA=1 AND
    OB(6)=−1) THENPRINT:PRINT "YOU
    CAN GO "; ELSE 46Ø
410 IF N>Ø THEN PRINT TAB(11);"NORTH"
420 IF E>Ø THEN PRINT TAB(11);"EAST"
430 IF S>Ø THEN PRINT TAB(11);"SOUTH"
440 IF W>Ø THEN PRINT TAB(11);"WEST"
```

The routine simply checks the value of the N, S, E, and W variables that you filled in, based on your map of the locations. If the value of the variables is greater than zero, then the direction is possible and the exit is displayed.

The routine can be incorporated as it stands, in any adventure based on a grid of squares.



INSTRUCTIONS?

Now that the adventurer knows which directions are possible, some kind of prompt ought to be given. The player will be asked WHAT NOW? by this section of program:

```
450 REM ** INSTRUCTIONS**
460 INPUT INVERSE 1;“WHAT NOW□”; LINE I$
470 GOSUB 3Ø1Ø
515 GOTO G(I,4)
```

```
450 REM ** INSTRUCTIONS **
460 PRINT: INPUT “■WHAT NOWπ”; I$
470 GOSUB 3Ø1Ø
```

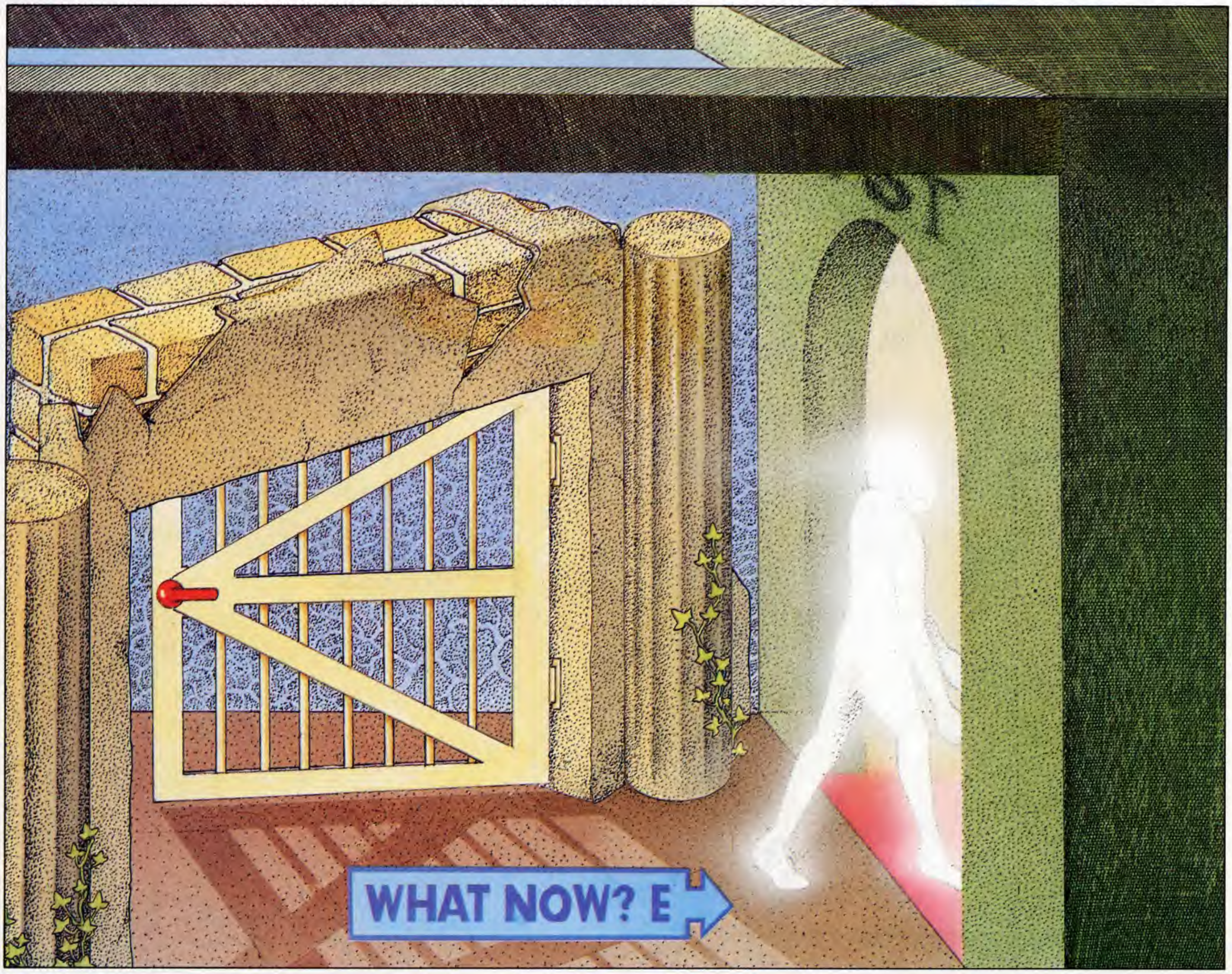
```
45Ø REM**INSTRUCTIONS**
46Ø INPUT “WHAT NOW”,I$
47Ø GOSUB 3Ø1Ø
```

```
45Ø REM **INSTRUCTIONS**
46Ø PRINT: INPUT “WHAT NOW”;I$
47Ø GOSUB 3Ø1Ø
```

In this very straightforward INPUT routine, the adventurer is asked WHAT NOW? and the response is called I$. The computer will then need to check the player's response and act upon it. The next line, Line 47Ø, sends the program on to a subroutine in Line 3Ø1Ø which handles the player's input.

```
6ØØ REM **INSTR ROUTINE**
61Ø LET IN = Ø: IF LEN Y$ > LEN X$ THEN
    RETURN
62Ø FOR K = 1 TO (LEN X$ – LEN Y$ + 1)
63Ø IF Y$ = X$(K TO K + LEN Y$ – 1) THEN
    LET IN = K: GOTO 65Ø
64Ø NEXT K
65Ø RETURN
3ØØØ REM **CHECK INSTRUCTION**
3Ø1Ø LET N$ = “”: LET X$ = I$: LET Y$ =
    “□”: GOSUB 6ØØ: LET I = IN
3Ø2Ø IF I = Ø THEN LET V$ = I$: GOTO 3Ø5Ø
3Ø3Ø LET V$ = I$(TO I – 1)
3Ø4Ø LET N$ = I$(I + 1 TO )
3Ø5Ø LET I = Ø
3Ø6Ø FOR K = 1 TO 19
3Ø7Ø IF V$ = R$(K, TO LEN V$) THEN LET
    I = R(K): LET I$ = V$( TO 1)
3Ø8Ø NEXT K
3Ø9Ø RETURN
```

```
3ØØØ REM ** CHECK INSTRUCTION **
3Ø1Ø N$ = “”:FOR Z = 1 TO LEN(I$):IF MID$
    (I$,Z,1) = “□” THEN I = Z:GOTO 3Ø2Ø
3Ø15 NEXT:I = Ø
3Ø2Ø IF I = Ø THEN V$ = I$:GOTO 3Ø5Ø
3Ø3Ø V$ = LEFT$(I$,I – 1)
3Ø4Ø N$ = MID$(I$,I + 1)
```



```
3Ø5Ø I=Ø
3Ø6Ø FOR K=1 TO 19
3Ø7Ø IF V$=LEFT$(R$(K),LEN(V$))
   THEN I=R(K):I$=LEFT$(V$,1)
3Ø8Ø NEXT
3Ø9Ø RETURN
```

```
3ØØØ REM**CHECK INSTRUCTION**
3Ø1Ø N$="":I=INSTR(I$,"□")
3Ø2Ø IF I=Ø THEN V$=I$:GOTO 3Ø5Ø
3Ø3Ø V$=LEFT$(I$,I-1)
3Ø4Ø N$=MID$(I$,I+1)
3Ø5Ø I=Ø
3Ø6Ø FOR K=1 TO 19
3Ø7Ø IF INSTR(R$(K),V$)=1
   THEN I=R(K):I$=LEFT$(V$,1)
3Ø8Ø NEXT
3Ø9Ø RETURN
```

The routine checks if I$ consists of two words. If it does, the first word is called V$, and the second, N$. V$ stands for verb, if you like—a word such as GET, KILL or CARRY, and all of the direction words such as NORTH, SOUTH, EAST and WEST. N$ stands for noun—the names of the objects in the adventure.

The Dragon, Tandy and Acorn machines use INSTR in Line 3Ø1Ø, to check for a space in I$—the break between N$ and V$. The Commodore, Vic and Spectrum, however, do not have INSTR. Instead, the Commodore program uses MID$ to search for the space—see Lines 3Ø1Ø and 3Ø15. On the Spectrum, though, there is a very useful little subroutine at Lines 6ØØ to 65Ø which impersonates the INSTR function on the other machines.

If a space is found, then Line 3Ø3Ø labels the two parts of I$ as N$ and V$. If no space is found, Line 3Ø2Ø relabels the whole of I$ as V$.

The remainder of the subroutine—Lines 3Ø6Ø to 3Ø8Ø—searches through the responses in R$. R$ is the array that currently holds the direction responses. Later on, you'll see how it can be expanded to hold a series of extra verbs. If Line 3Ø7Ø finds a match, then I is set to the corresponding value from R. Later in the program, the machine will know if a match has been found by checking if I is greater than zero. The last part of Line 3Ø7Ø takes the first letter of V$ and calls it I$. I$ will be used later for moving the adventurer around.

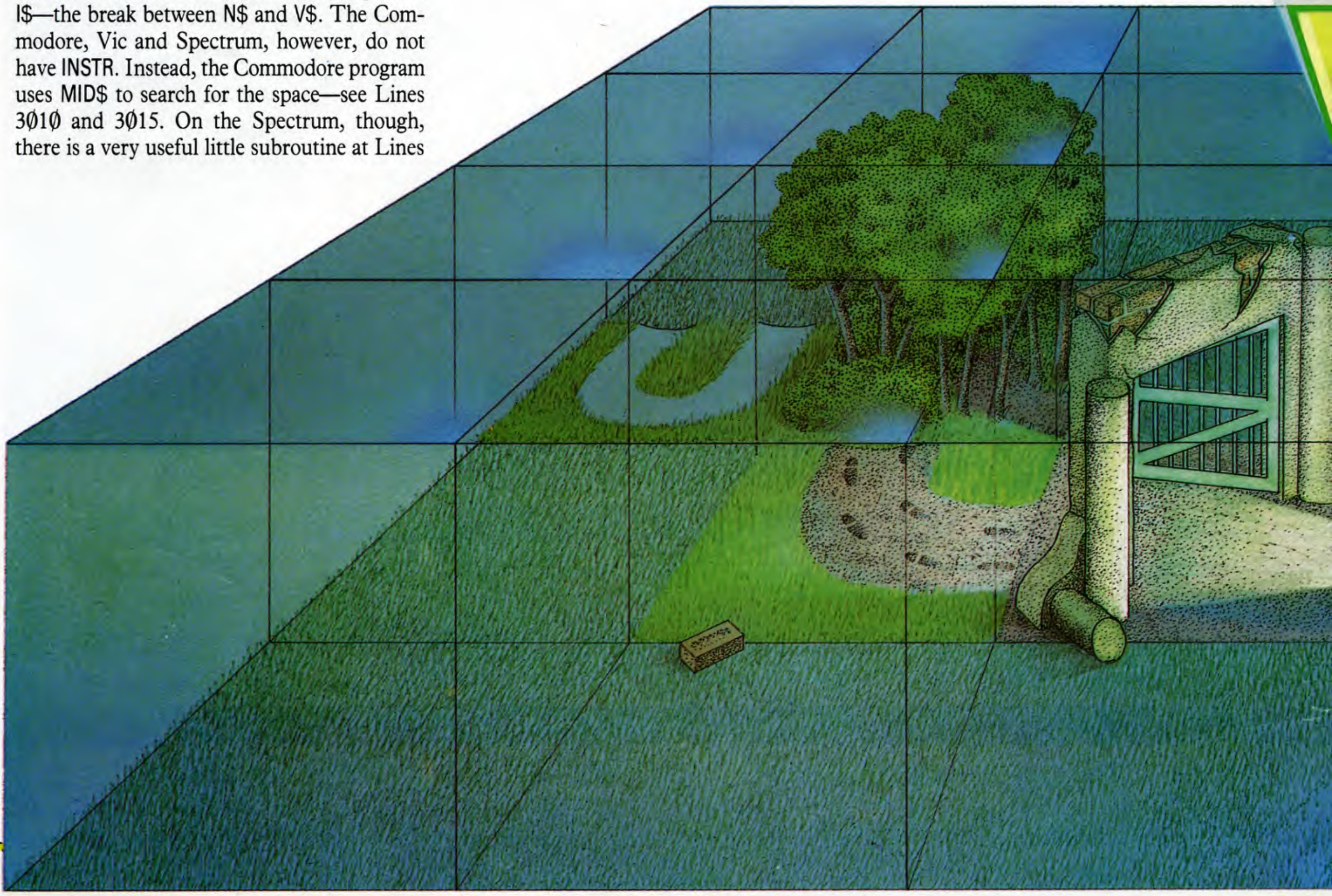
The subroutines can be used almost without alteration in any adventure game. There is only one place adjustments may have to be made—in the length of the FOR ... NEXT loop in Line 3Ø6Ø.

MOVING AROUND

All you need to add so that the adventurer can explore all the locations is a routine which manipulates the location variable L, according to I$. Here it is:

```
1ØØØ REM **MOVEMENT ROUTINE**
1Ø1Ø IF I$="N" AND N>Ø THEN LET
   L=L-6: GOTO 31Ø
1Ø2Ø IF I$="E" AND E>Ø THEN LET
   L=L+1: GOTO 31Ø
1Ø3Ø IF I$="S" AND S>Ø THEN LET
   L=L+6: GOTO 31Ø
1Ø4Ø IF I$="W" AND W>Ø THEN LET
   L=L-1: GOTO 31Ø
1Ø5Ø REM **IF NO LOCATION POSSIBLE IN
   DIRECTION**
1Ø6Ø PRINT "'SORRY YOU CAN'T GO THAT
   WAY !!": GOTO 33Ø
```

```
1ØØØ REM **MOVEMENT ROUTINE**
1Ø1Ø IF I$="N" AND N>Ø THEN
   L=L-6:GOTO 31Ø
```



```
1Ø2Ø IF I$=“E” AND E>Ø THEN
   L=L+1:GOTO 31Ø
1Ø3Ø IF I$=“S” AND S>Ø THEN
   L=L+6:GOTO 31Ø
1Ø4Ø IF I$=“W” AND W>Ø THEN
   L=L−1:GOTO 31Ø
1Ø5Ø REM **IF NO LOCATION POSSIBLE IN
   DIRECTION**
1Ø6Ø PRINT:PRINT “SORRY —YOU CAN’T GO
   THAT WAY !!”:GOTO 33Ø
```

```
1ØØØ REM**MOVEMENT**
1Ø1Ø IF I$=“N” AND N>Ø THEN
   L=L−6:GOTO 31Ø
1Ø2Ø IF I$=“E” AND E>Ø THEN
   L=L+1:GOTO 31Ø
1Ø3Ø IF I$=“S” AND S>Ø THEN
   L=L+6:GOTO 31Ø
1Ø4Ø IF I$=“W” AND W>Ø THEN
   L=L−1:GOTO 31Ø
1Ø5Ø REM****CAN’T GO THERE****
1Ø6Ø PRINT “‘SORRY—YOU CAN’T GO THAT
   WAY !!”:GOTO 33Ø
```

You’ll remember that the adventure was based on a grid six locations wide. Moving around the adventure involves altering the value of L by a factor based on the grid size. Moving North and South involves adding and subtracting 6 from L to drop you or raise you by one whole line on the grid. Similarly, moving East and West involves adding or subtracting 1 from L.

Lines 1Ø1Ø to 1Ø4Ø check I$ for direction instructions and adjust L appropriately. Movement will only be possible if there is an exit which matches I$. The exits were defined in the lines immediately following the location descriptions.

If there isn’t an exit in the direction that the adventurer wants to go, Line 1Ø6Ø displays the message SORRY—YOU CAN’T GO THAT WAY!!

The only alteration you will have to make if you want to use the routine in a different adventure will be if the grid is a different size. In that case, Lines 1Ø1Ø and 1Ø3Ø will have to be altered according to the width of the grid.

Now SAVE the program ready for the next part of Games Programming.

Q+A

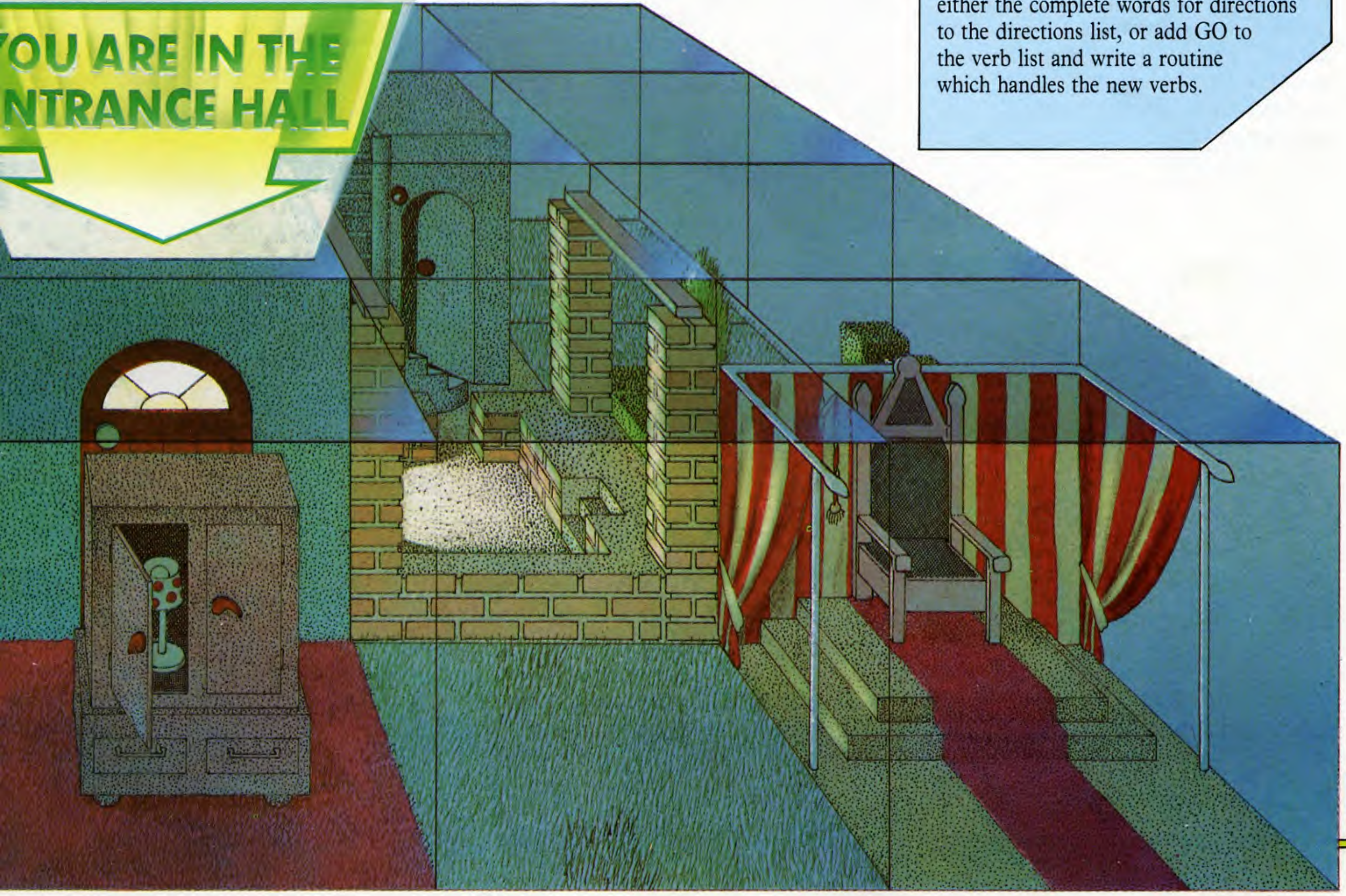
Will the program understand direction instructions such as NORTH and GO NORTH as well as single letters such as N?

The Check Instruction routine—from Line 3ØØØ to Line 3Ø1Ø—is specially written so that it picks out single letter instructions and handles them separately from longer instructions.

The single letter instructions are called I$ and later on the program will check I$ for the letters N, S, E and W, allowing the player to type the shortest form of the instruction, and making the game quicker to play

There is nothing to stop you making additions which will allow the adventurer to use NORTH or GO NORTH, though. Next time you will see how the program handles verbs and nouns.

What you should do is to add either the complete words for directions to the directions list, or add GO to the verb list and write a routine which handles the new verbs.



DRAGON/TANDY ANIMATED GRAPHICS

Find out how to use the GET and PUT commands on the Dragon and Tandy. Once you've mastered them it's simple to animate almost any graphic

Once you can set up both black and white and colour UDGs you will almost certainly want to move them around the screen. To do this, you need to use the GET and PUT commands which you have already seen used many times in Games Programming—see page 144, for example.

GET can be thought of as being similar to taking a photograph of an area of the Dragon's high resolution screen. Whatever is being displayed in that area will be stored in an array (see Basic Programming, page 152)—it doesn't matter if the area contains a UDG or a graphic displayed using some other graphics command such as LINE, CIRCLE or DRAW. You can GET any rectangular area, up to and including the whole screen.

PUT is the reverse of GET, and literally puts the photograph—the contents of the array—back on the screen anywhere you wish. Using PUT is much quicker than re-drawing the graphic using other methods, and animation can easily be simulated by repeatedly PUTting an image in different positions, or by repeatedly PUTting different images.

ON YOUR BIKE

Type in and RUN this program which demonstrates using GET and PUT to animate a UDG of a bicycle:

```
10 PMODE 4,1:PCLS
20 DIM BY(5)
30 FOR K=1536 TO 1856 STEP 32
40 READ A,B,C
50 POKE K,A:POKE K+1,B:POKE K+2,C
60 NEXT
70 SCREEN 1,1
80 GET (1,0)-(18,11),BY,G
90 PCLS
100 FOR K=0 TO 238
110 PUT(K,90)-(K+17,100),BY,PSET
120 NEXT
130 GOTO 90
140 DATA 1,193,192,0,129,32,0,255,
    64,1,134,0,14,139,128,19,86,64
150 DATA 36,233,32,47,233,32,32,104,
    32,17,100,64,14,3,128
```

The DATA from Lines 140 and 150 is called up and POKEd on the screen by Lines 30 to 60. The bicycle is drawn in the top left-hand corner, or the beginning of the screen. Whenever you want to POKE a graphic on the screen and then use GET and PUT it's a good idea to draw the graphic here first because you know exactly where the graphic is—converting memory locations into screen locations isn't particularly easy because of variations between PMODEs. Exactly how this works is explained a little later. Line 80 GETs (or 'takes a photograph', if you like, of) the bicycle.

- SETTING UP A UDG AND ANIMATING IT
- USING ARRAYS WITH THE **GET** AND **PUT** COMMANDS
- **PUT**TING WHERE YOU WISH
- DIMENSIONING THE ARRAYS CORRECTLY
- 'TAKING PHOTOGRAPHS' OF THE SCREEN
- ANIMATION TECHNIQUES

The screen is cleared by Line 9Ø before Lines 1ØØ to 12Ø animate the bicycle. Line 11Ø PUTs the bicycle on the screen at a new position each time through the FOR ... NEXT loop.

DIMENSIONING THE ARRAYS

Whenever you want to use GET in a program you'll have to DIMension an array in which to store the graphics information. To work out the size of array needed, follow these steps:

1. Once you have designed your graphic, preferably on graph paper, draw a rectangle which completely encloses it. Count how many *individual* pixels there are along the top of the rectangle, and how many up the side. Now multiply the two numbers together. This gives you the number of pixels that the graphic occupies. For example, the rectangle which will enclose the bicycle is 18 pixels wide, by 12 pixels high. Multiplying them together, there are 18*12 pixels—216 in all.

2. Next you need to work out how many bytes in memory are needed to store that number of pixels. In PMODEs 3 and 4 you'll have to divide the number of pixels by 8, in PMODEs 1 and 2 you'll have to divide by 16, and in PMODE Ø, by 32. As the bicycle has been drawn in PMODE 4 you'll have to divide 216 by 8—giving the answer 27. The number you use in the calculation has to be a *whole number* of bytes, so this answer may have to be adjusted. Just round it up to the next whole number. No matter what the size of the decimal, always round up, not down.

3. Work out the size of array you need to DIMension by dividing the number of bytes by 5. Always divide by 5 at this stage, no matter which PMODE is being used. To return to the example of the bicycle: there are 27 bytes to be stored in the array, so 27/5 = 5.4. Again, if the result of the division isn't an integer, you must round up. For the bicycle, an array with 6 elements is needed—Line 2Ø says DIM BY(5) because arrays start at element Ø.

Here is a summary of the divisors you'll need in order to work out the sizes of arrays in your graphics programs:

	To work out number of:	
PMODE	Bytes	Arrays
4	8	5
3	8	5
2	16	5
1	16	5
Ø	32	5

GET can be used with any type of graphics on the computer's high resolution screen no matter which way they were created. It doesn't matter whether you've set up a UDG, or used PSET and PRESET, or DRAW—or any combination of the graphics commands—you can still use GET to store the pictures in arrays for PUTting back on the screen later.

To GET a graphic into any array you have DIMensioned you must tell the computer where to find the graphic on the screen, and in which array you wish it to be stored. Look at this line from the bicycle program:

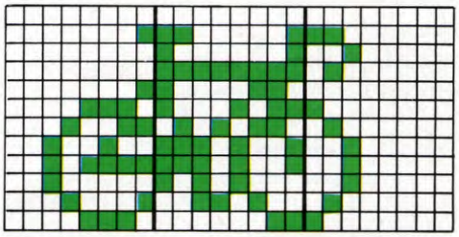
```
8Ø GET(1,Ø)–(18,11),BY,G
```

The numbers in the brackets are the screen coordinates of diagonally-opposite corners of the rectangle which encloses the graphic. It doesn't matter which corners are specified, nor in what order, but they *must* be diagonally-opposite corners.

Notice that the area which has been stored is not the whole UDG which you defined, but only the part of it which actually contains the images—because the bicycle doesn't occupy the whole 24-pixel-wide block (three 8-pixel UDGs). Using GET on the computer doesn't limit you just to moving multiples of 8×8 pixels as on some other computers so you can animate almost anything you like.

The next part of Line 8Ø after the screen location, BY, tells the computer to store the graphic in array BY, which has been DIMensioned in Line 2Ø.

Finally, G stands for 'full Graphics detail'. You can omit G in certain circumstances (which will be explained a little later) but is generally advisable to leave it in.



If you are setting up graphics as UDGs it's a very good idea to POKE them into the top left hand corner of the screen, as explained before, because it makes the GET statement easier to set up. Remember that when you POKE on to the screen, the location of the image is not contained in ordinary screen coordinates. But because you have to GET the image from particular screen coordinates you will have to convert the memory locations you've POKEd into, to the screen coordinate equivalent. It's not very easy to do this conversion because it will vary between PMODEs. POKEing into memory locations at the start of the screen—beginning at location 1536—makes the screen location of the UDG very easy to work out—the top left hand corner must be at coordinates (Ø,Ø) and since you know the size of the graphic, it's easy to work out the coordinates of the bottom right hand corner of the UDG when you GET it.

USING PUT

Once you used GET to store the graphic in memory you should clear the screen using PCLS before you PUT the graphic back on the screen.

Here's the PUT line for the bicycle program:

```
11Ø PUT(K,9Ø) — (K+17,1ØØ),BY,PSET
```

The figures in the brackets, just as with GET, give the coordinates of two diagonally-opposite corners which define the area in which you want to PUT the image. BY is the array which now contains the bicycle.

The last part of the line is PSET. PSET tells the computer to PUT the graphic on the screen exactly as it was originally drawn, obliterating whatever is already at the screen location you desire.

There are other options, though. PSET can be replaced by PRESET, OR, AND or NOT, which allow you to manipulate the graphic.

PRESET tells the computer to PUT the graphic on the screen in inverse form—in a two-colour mode, the two colours will simply reverse. In a four-colour mode used with colour set Ø, red will become green, and blue will become yellow, and vice versa. In colour set 1, orange will become buff, and cyan will become magenta, and vice versa.

OR allows you to superimpose the graphic stored in the array over whatever is already on the screen. Both graphics will remain unaltered, unlike PSET, where the original graphic would have been obliterated.

The use of AND and NOT may seem a little obscure. AND displays only the overlap between a graphic already on the screen and the graphic which is being PUT onto that location. NOT doesn't display any of the contents of the array. All it does is to reverse the area on the screen that the array is PUT on. To see why you might need these commands, type in and RUN the following program:

```
1Ø PMODE 4,1
2Ø DIM C(5),B1(5),B2(5)
3Ø PCLS
4Ø CIRCLE (7,7),7,1:PAINT(7,7),1,1
5Ø GET (Ø,Ø) — (14,14),C,G
6Ø PUT (Ø,Ø) — (14,14),C,NOT
7Ø GET (Ø,Ø) — (14,14),B2,G
8Ø PCLS1:SCREEN 1,Ø
9Ø LINE(8,191) — (1Ø4,Ø),PRESET:
   LINE(24,191) — (12Ø,Ø),PRESET:
   PAINT(1Ø,191),Ø,Ø
1ØØ LINE(1Ø4,191) — (2ØØ,Ø),PRESET:
   LINE(12Ø,191) — (216,Ø),PRESET:
   PAINT(11Ø,191),Ø,Ø
11Ø FOR K=175 TO Ø STEP —1Ø
12Ø X=14+(175—K)/2
13Ø PUT(X,K) — (X+14,K+14),C,OR
14Ø PUT(X+96,K) — (X+11Ø,K+14),C,OR
15Ø FOR J=1 TO 1ØØ:NEXT
16Ø PUT(X,K) — (X+14,K+14),B1,PSET
17Ø PUT(X+96,K) — (X+11Ø,K+14),B2,
   AND
18Ø NEXT
19Ø GOTO 8Ø
```

For an explanation of the method used for the animation effect, see below.

If you look at what happens as the two balls travel from bottom to top of the screen you'll see some 'corners' appearing along the side of the black stripes. The reason for this is that you are trying to fit a rectangular graphic into a slanted one. And every time the program tries to PUT a block graphic on the screen to blank out the ball's last position, the graphic overlaps the stripe, creating the corners.

The ball travelling up the right hand stripe causes no such problems. Using a combination of PUT . . . , AND and PUT . . . , OR the corners can be eliminated. First the ball is reversed on the screen by Line 6Ø, and then it is stored in memory by Line 7Ø. When the program animates the ball, Line 14Ø PUTs a blank on the screen on top of the stripe. This is a PUT . . . , OR so the last ball is wiped out by a precisely-shaped blank. The reversed ball is PUT on the screen using PUT . . . , AND. The combination of NOT with AND allows the ball to be PUT on the screen by Line 17Ø with no overlap.

If you want to PUT a graphic on the screen in a non-rectangular space—a circle or a triangle in a circular or traingular hole, perhaps—you can use this method. First draw the object. PUT it back on the screen using PUT . . . , NOT and then GET it into an array. When you want to display the graphic on the screen later, use PUT . . . , AND.

WHEN TO LEAVE OUT G

Earlier you read that it was possible to omit G from a GET statement. To understand when G can be omitted you will have to know more about the computer's screen display.

In PMODEs 1, 3 and 4 the graphics screen is divided into 32 vertical columns, each eight pixels wide, whilst in PMODEs Ø and 2 the screen is divided into 16 columns, each 16 pixels wide. In whichever PMODE you use, the width of the screen columns each correspond to one byte in memory.

If the graphic you have designed occupies an exact number of columns and you want to move it a precise number of columns, you can omit G from the GET statement. Without G, *whole* bytes are stored in the array you've named. If your graphic overlaps parts of columns you will be needing to store *parts* of bytes in the array. In other words, you need some way of storing *bits* from the screen in the array. This is the function of G.

If you leave out G when your graphic overlaps screen columns a corrupted version of your original graphic will be PUT on the screen. If you have correctly omitted G then you *must* also omit the PSET, PRESET, OR, AND or NOT from the corresponding PUT. Omitting G, then, will not allow you such great flexibility with PUT.

ANIMATION USING GET AND PUT

GET and PUT are probably most useful when you want to write a program using animated graphics. You can use two main methods:

First, you may use an empty array, a blank, to blot out the graphic before PUTting it back on the screen somewhere else. Make sure that the blank is the same size as your graphic and you should have no problems. The technique has been used a number of times in Games Programming (see page 144, for example).

The second method, used in the bicycle animation, uses a blank border of pixels round the graphic. First decide how many pixels the graphic is going to move in each step, and then make sure that there are that number of rows of blank pixels on the trailing edge of the graphic. This means that the new graphic PUT on the screen blanks out the previous graphic. If the graphic moves from side to side and up and down, say four pixels at a time, you need a border of four rows of blank pixels surrounding the graphic. In the case of the bicycle, the graphic is moved in one-pixel steps from left to right across the screen, so a single row of blank pixels had to be left along the left hand side of the graphic when it was stored.

Finally, if you think that this has been rather a lot to absorb, try designing some graphics. Use GET and PUT to draw them elsewhere on the screen, or animate them. The most effective way to get to grips with PSET, PRESET, AND, NOT and OR is to experiment.

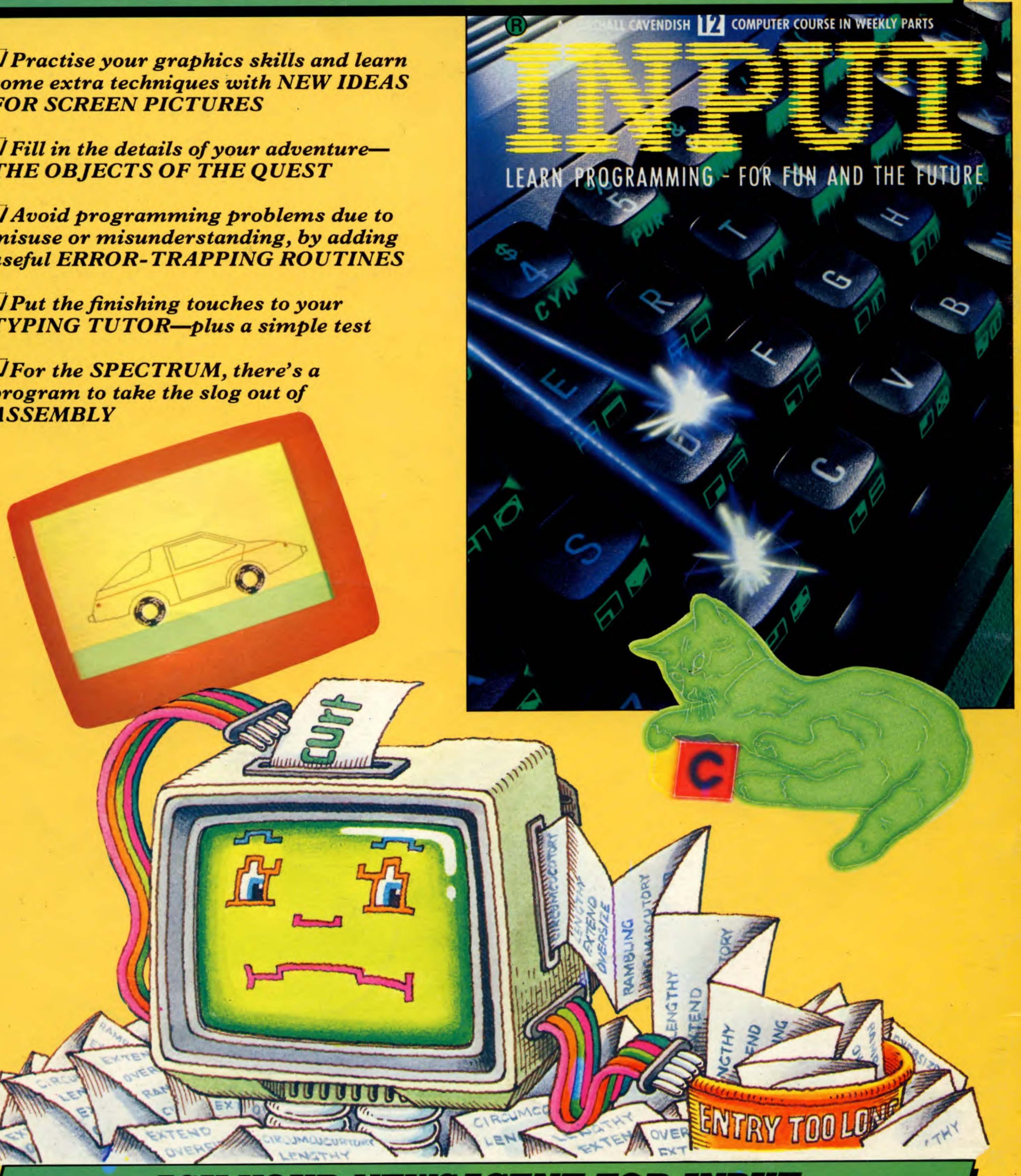
CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT.*

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 12...

- ❑ *Practise your graphics skills and learn some extra techniques with NEW IDEAS FOR SCREEN PICTURES*
- ❑ *Fill in the details of your adventure—THE OBJECTS OF THE QUEST*
- ❑ *Avoid programming problems due to misuse or misunderstanding, by adding useful ERROR-TRAPPING ROUTINES*
- ❑ *Put the finishing touches to your TYPING TUTOR—plus a simple test*
- ❑ *For the SPECTRUM, there's a program to take the slog out of ASSEMBLY*


A MARSHALL CAVENDISH 12 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE
C
CURT
CIRCUMLOCUTORY
LENGTHY
EXTEND
OVERSIZE
RAMBLING
ENTRY TOO LONG

ASK YOUR NEWSAGENT FOR INPUT